

ADVANCING MEMORY-CORRUPTION
ATTACKS AND DEFENSES

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

M.Sc. Christopher Liebchen

Referenten:

Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)

Prof. Dr. Thorsten Holz (Zweitreferent)

Tag der Einreichung: 27. Februar 2018

Tag der Disputation: 9. Mai 2018



TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Christopher Liebchen:

Advancing Memory-corruption Attacks and Defenses, © February 2018

PHD REFEREES:

Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st PhD Referee)

Prof. Dr. Thorsten Holz (2nd PhD Referee)

FURTHER PHD COMMISSION MEMBERS:

Prof. Dr. Dr. h.c. Johannes Buchmann

Prof. Dr. Sebastian Faust

Prof. Dr.-Ing. Mira Mezini

Darmstadt, Germany February 2018

Veröffentlichung unter CC-BY-NC-ND 4.0 International

<https://creativecommons.org/licenses/>

ABSTRACT

Adversaries exploit software vulnerabilities in modern software to compromise computer systems. While the amount and sophistication of such attacks is constantly increasing, most of them are based on memory-corruption vulnerabilities—a problem that has been persisting over the last four decades. The research community has taken on the challenge of providing mitigations against memory-corruption-based attack techniques such as code-injection, code-reuse, and data-only attacks. In a constant arms race, researchers from academia and industry developed new attack techniques to reveal weaknesses in existing defense techniques, and based on these findings propose new mitigation techniques with the goal to provide efficient and effective defenses in the presence of memory-corruption vulnerabilities. Along this line of research, this dissertation contributes significantly to this goal by providing attacks on the recently proposed mitigations and more enhanced defenses against memory-corruption-based attacks.

Specifically, we present sophisticated attacks against the Control-flow Integrity (CFI) implementation of two premier open-source compilers, and demonstrate conceptual limitations of coarse- and fine-grained CFI. Our first attack exploits a compiler-introduced race-condition vulnerability, which temporarily spills read-only CFI-critical variables to writable memory, and hence, enables the attacker to bypass the CFI check. Our second attack is a data-only attack that modifies the intermediate representation of the Just-in-Time (JIT) compiler in browsers to generate attacker-controlled code. We then turn our attention to attacking randomization-based defenses. We demonstrate conceptual limitations of randomization with two advanced memory-disclosure attack techniques. In particular, we demonstrate that the attacker can bypass any code-randomization either by reading the code directly, or indirectly by combining static code analysis with a sufficient number of disclosed code pointers.

Based on the insights we gain through our attack techniques, we design and implement a leakage-resilient code randomization scheme to defeat code-reuse attacks by using execute-only memory to mitigate memory-disclosure attacks. Since x86 does not natively support execute-only memory, we leverage memory virtualization to enable it for server and desktop systems. Moreover, since most embedded systems do not offer memory virtualization, we demonstrate how to overcome this limitation by implementing a compiler extension that enables software-based execute-only memory for ARM-based systems. Lastly, we demonstrate how leakage-resilient randomization can also be deployed to mitigate data-only attacks against the page table.

ZUSAMMENFASSUNG

Angreifer nutzen Programmierfehler in Software aus, um verwundbare Computersysteme zu kompromittieren. Während sowohl die Anzahl, als auch die Komplexität dieser Angriffe weiterhin zunimmt, hat sich an der zugrundeliegenden Ursache nichts geändert: Seit mehr als vier Jahrzehnten nutzen Angreifer Speicherfehler aus, um den Kontroll- oder Datenfluss des Programms zur Laufzeit zu manipulieren. Aus diesem Grund haben es sich Forscher in Universitäten und Unternehmen zum Ziel gesetzt, effektive und effiziente Verteidigungstechniken gegen speicherfehlerbasierte Angriffe zu entwickeln. Mit dieser Dissertation tragen wir maßgeblich zu diesem Ziel bei, indem wir neue Angriffstechniken entwickeln und, basierend darauf, neue Verteidigungstechniken entwerfen.

Im Besonderen zeigen wir Schwächen bei der Umsetzung feingranularer Kontrollflussintegrität in zwei weitverbreiteten Compilern sowie konzeptionelle Schwächen von grob- und feingranularer Kontrollflussintegrität im Allgemeinen auf. Unsere erste Angriffstechnik nutzt eine Wettlaufsituation aus, die ungewollt vom Compiler durch die Optimierung des generierten Programmcodes und der darin enthaltenen Kontrollflussintegritätsverifikationen eingefügt wird. Dabei werden Werte, die zur Überprüfung der Kontrollflussintegrität aus nicht-schreibbaren Speicher in Register geladen wurden, während eines Funktionsaufrufes temporär in schreibbaren Speicher zwischengespeichert. Dort können diese Werte manipuliert und die schützende Kontrollflussintegrität umgangen werden. Unsere zweite Angriffstechnik modifiziert die verwendete Zwischendarstellung des Laufzeitcompilers eines Webbrowsers, wodurch dieser Schadcode generiert. Neben den integritätsbasierten Verteidigungstechniken überprüfen wir auch randomisierungsbasierte Verteidigungstechniken auf deren Sicherheit. In diesem Zusammenhang entwickeln wir zwei fortgeschrittene, auf Speicherlecks basierende Angriffstechniken, die die konzeptionellen Schwächen von randomisierungsbasierten Verteidigungstechniken verdeutlichen.

Basierend auf den Erkenntnissen, die wir durch das Entwickeln genannter Angriffstechniken gewonnen haben, entwerfen und implementieren wir eine Technik, um randomisierungsbasierte Verteidigungstechniken vor Speicherlecks zu schützen. Diese basiert auf nur-ausführbarem Speicher, welchen wir auf der x86 Architektur mittels Speichervirtualisierung ermöglichen. Weiter zeigen wir, dass nur-ausführbarer Speicher für eingebettete Systeme, welche oft keine Speichervirtualisierung unterstützen, mit Hilfe einer Compilererweiterung auch ohne jegliche Hardwareunterstützung umgesetzt werden kann. Zuletzt entwerfen wir eine gegen Speicherlecks resistente, randomisierungsbasierte Verteidigung, die die Datenstruktur zur Verwaltung des virtuellen Speichers vor datenbasierten Angriffen schützt.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Prof. Ahmad-Reza Sadeghi for the opportunity to pursue my PhD at his research group. Throughout the years he provided guidance, feedback, and discussions that had a significant impact on my PhD studies. His dedication to security research is exceptional, and was a constant source of motivation. I am deeply grateful for the opportunities he gave to me, and especially for establishing collaborations with outstanding security researchers worldwide.

Besides Ahmad, I would like to thank Prof. Lucas Davi from Universität Duisburg-Essen with whom I had the privilege to collaborate on several projects while he was still a PhD student at Ahmad's group. Further, I would like to thank Prof. Fabian Monrose from the University of North Carolina at Chapel Hill who provided constructive feedback at the beginning of my PhD. I thank Johannes Buchmann, Sebastian Faust, and Mira Mezini for agreeing to join the PhD commission.

During my PhD I had the honor to collaborate with excellent security researches. In particular, I would like to thank Stephen Crane, Per Larsen, Andrei Homescu and Prof. Michael Franz for the close collaboration and fruitful discussion. Additionally, I would like to thank my co-authors Alexandra Dmitrienko, Bjorn De Sutter, Christian Rossow, Daeyoung Kim, David Bigelow, David Gens, Dean Sullivan, Felix Schuster, Ferdinand Brasser, Georg Koppen, Hamed Okhravi, Kevin Snow, Liviu Iftode, Marco Negro, Mauro Conti, Mike Perry, Mohaned Qunaibit, Orlando Arias, Richard Skowyra, Robert Rudd, Stefan Brunthaler, Stijn Volckaert, Thomas Hobson, Thomas Tendyck, Thorsten Holz, Tommaso Frassetto, Veer Dedhia, Vinod Ganapathy and Yier Jin for all their hard work. Special credits go to Ferdinand Brasser, Stephan Heuser, and Alexander Frömmgen for their critical discussion on several research ideas and other topics, and to all my colleagues from Ahmad's group.

Over the years I was fortunate to supervise a number of theses of talented students. With Kjell Braden we implemented in his master thesis techniques to enable execute-only memory for embedded devices. In David Gens' master thesis, we developed a randomization-based defense to mitigate data-only attacks against page tables. For Tommaso Frassetto's master thesis we demonstrated a scheme for highly practical load-time code randomization. Markus Schader's bachelor thesis provided some initial results for a payload for data-only attacks against JIT compilers. With David Rieger's master thesis we explored the possibility of using multi-architecture execution to mitigate code-reuse attacks, and in Patrick Jauernig's master thesis we designed and implemented lightweight in-process isolation for the x86 architecture.

In the last year of my PhD I got the opportunity to do an internship at Qualcomm in San Diego, California. I would like to thank my manager Pouyan Sepehrdad, my mentor Akash Waran and Daniel Godas-Lopez for their advice and support, and the rest of the Product Security team for the interesting discussions.

CONTENTS

1. Introduction	1
1.1 Goals and Scope of this Dissertation	2
1.2 Summary of Contribution	3
1.3 Outline	5
1.4 Previous Publications	5
2. Background	9
2.1 Low-level View of an Application	9
2.2 Memory-corruption Attacks	12
2.3 Code-injection Attacks and Defenses	14
2.4 Code-reuse Attacks and Defenses	16
3. Advances in Memory-Corruption Attacks	25
3.1 Memory-disclosure Attacks	25
3.1.1 Threat Model	26
3.1.2 Direct-disclosure Attacks	27
3.1.3 Indirect-disclosure Attacks	29
3.1.4 Conclusion	34
3.2 Attacks on Fine-grained CFI	35
3.2.1 Threat Model	36
3.2.2 StackDefiler	37
3.2.3 Attack Implementations	40
3.2.4 Mitigations	46
3.2.5 Discussion	49
3.2.6 Conclusion	50
3.3 Attacks on Coarse-grained CFI	51
3.3.1 Background on Coarse-grained CFI and C++	51
3.3.2 Counterfeit Object-oriented Programming	53
3.3.3 Evaluation	56
3.3.4 Conclusion	57
3.4 Data-only Attack on JIT compilers	58
3.4.1 Background on SGX and JIT Compilation	59
3.4.2 Threat Model	62
3.4.3 Our Data-only Attacks on JIT Compilers	63
3.4.4 Conclusion	66
3.5 Related Work	68
3.5.1 Attacks against Control-flow Integrity	70
3.5.2 Data-only Attacks	71
3.6 Summary and Conclusion	73

4. Advances in Memory-Corruption Defenses	75
4.1 Readactor: Memory-Disclosure Resilient Code Randomization	75
4.1.1 Threat Model	77
4.1.2 Readactor Design and Implementation	78
4.1.3 Security Evaluation	84
4.1.4 Performance Evaluation	87
4.1.5 Discussion	89
4.1.6 Conclusion	90
4.2 LR ² : Software-based Execute-only Memory	91
4.2.1 Threat Model	92
4.2.2 LR ²	93
4.2.3 Performance Evaluation	103
4.2.4 Register-Register Addressing Scheme Restrictions	104
4.2.5 Security Analysis	106
4.2.6 Discussion and Extensions	110
4.2.7 Conclusion	113
4.3 Selfrando: Practical Load-time Randomization	114
4.3.1 Design and Implementation	114
4.3.2 Evaluation	116
4.3.3 Conclusion	118
4.4 PT-Rand: Mitigating Attacks against Page Tables	119
4.4.1 Background on Memory Protection and Paging	121
4.4.2 On the Necessity of Page Tables Protection	123
4.4.3 Threat Model	126
4.4.4 Overview of PT-Rand	127
4.4.5 Implementation and Evaluation	128
4.4.6 Discussion	135
4.4.7 Conclusion	135
4.5 Related Work	137
4.5.1 Leakage-Resilient Diversity	137
4.5.2 Integrity-based defenses	139
4.5.3 Data-only Defenses	143
4.5.4 Kernel and Page-Table Attack Mitigations	143
4.6 Summary and Conclusion	145
5. Discussion and Conclusion	147
5.1 Dissertation Summary	147
5.2 Future Research Directions	148
6. About the Author	151
Bibliography	155

LIST OF FIGURES

1	High-level memory layout and access permissions of an application during run time.	10
2	Stack frames on x86.	11
3	During a buffer overflow the attacker writes past the allocated buffer bounds.	13
4	Code-injection Attack	14
5	Stack layout during a code-injection attack.	15
6	Code-reuse Attack	16
7	Return-oriented programming attack	17
8	Attacker exploits a heap-based buffer overflow and leverages a stack pivot gadget to launch a return-oriented programming (ROP) attack.	18
9	Example of unaligned instructions on x86 64-bit.	19
10	Address Space Layout Randomization changes the base address of code and data sections.	20
11	Control-flow Integrity verifies the target of indirect branches before executing them.	22
12	Static verification of return targets is too imprecise.	22
13	Control-flow Integrity (CFI) can leverage a shadow stack to enforce that return instructions only return to the call site that invoked the current function.	23
14	Direct and indirect memory disclosure.	26
15	Detailed workflow of a Just-in-Time Return-oriented Programming (JIT-ROP) attack.	27
16	Example of how disclosing a virtual table (vtable) pointer allows the attacker to identify valid mapped code pages.	30
17	Heap-Layout of our Exploit.	32
18	Application compiled with position-independent code. To get the absolute address of <code>str</code> the compiler emits instructions that first receive the absolute address of <code>Function</code> at run time. The absolute address of <code>str</code> is then calculated by adding the relative offset between <code>Function</code> and <code>str</code> , calculated by the compiler, to the absolute address of <code>Function</code>	38
19	The attacker can overwrite the length field of an array object. He uses the native read function to disclose memory content beyond the array buffer, e.g., the vTable pointer of a consecutive object.	41
20	SPEC CPU2006 performance of IFCC-protected programs before and after we applied our fix relative to an unprotected baseline.	47
21	Memory representation of C++-memory objects.	53

22	Process of chaining Counterfeit Object-oriented Programming (COOP) gadgets.	54
23	Concept of overlapping C++objects.	56
24	Main components of a JavaScript JIT engine.	59
25	During JIT spraying the attacker exploits that large constants are directly transferred into the native code. By jumping into the middle of an instruction the attacker can execute arbitrary instructions that are encoded into large constants.	61
26	DOJITA enables the attacker to execute arbitrary code through a data-only attack. In particular, the attacker manipulates the IR which is then used by the JIT compiler to generate native code that includes a malicious payload.	63
27	The IR of ChakraCore consists of a linked list of IR:Instr C++objects. The attacker injects instructions by overwriting the m_next pointer of a benign object (dotted line) to point to a linked list of crafted objects.	65
28	System overview. Our compiler generates diversified code that can be mapped with execute-only permissions and inserts trampolines to hide code pointers. We modify the kernel to use EPT permissions to enable execute-only pages.	78
29	Relation between virtual, guest physical, and host physical memory. Page tables and the EPT contain the access permissions that are enforced during the address translation.	80
30	Readactor creates two mappings for each physical memory page: a readacted mapping, which maps the physical memory as execute-only, and a normal mapping which maps the physical memory as read-write-execute. The operating system can map individual pages as execute-only by mapping virtual memory of a process either to the normal or readacted guest physical memory page.	82
31	Readacted applications replace code pointers in readable memory with trampoline pointers. The trampoline layout is not correlated with the function layout. Therefore, trampoline addresses do not leak information about the code to which they point.	83
32	In readacted applications, the function pointer tables are substituted with trampolines. Further, their entries are randomized, and, to counter brute-force attack on the entropy of the table layout, we insert trampolines to trap functions.	84
33	Performance overhead for SPEC CPU2006 with Readactor enabled relative to an unprotected baseline build.	88

34	Left: In legacy applications, all pages are observable by attackers. The stack, heap and global areas contain pointers that disclose the location of code pages. Right: In LR ² applications, attackers are prevented from observing the upper half of the address space which contains all code. Moreover, attacker observable memory only contains trampoline pointers (dotted arrows) that do not disclose code locations. Finally, return addresses on the stack are encrypted (not shown).	94
35	Differences between load-masking for software-fault isolation (left) and software-enforcement of XoM (right). Because SFI must consider existing code malicious, it must mask load addresses directly before every use. In contrast, software XoM is protecting trusted code executing legitimate control-flow paths, and can therefore use a single masking operation to protect multiple uses.	97
36	LR ² overhead on SPEC CPU2006. We use the performance of unprotected position independent binaries as the baseline.	103
37	Comparing software XoM to SFI (NaCl) to quantify effect of load-mask optimization.	106
38	Simplified disassembly of the function <code>v8::internal::ElementsAccessorBase::Get</code> that is used to read arbitrary memory. The load instruction in line 12 reads the memory from the base address provided in register <code>r1</code> plus the offset in register <code>r2</code> . After the instrumentation, this load is restricted by masking the MSB (line 11) which prevents reads into the code segment.	110
39	Workflow of selfrando.	115
40	Paging - translation of virtual addresses to physical addresses.	122
41	Overview of the different components of PT-Rand.	127
42	The x86_64 virtual memory map for Linux with four level page tables.	129
43	Probability for guessing attacks based on the number of mapped pages in the PT-Rand region.	131

LIST OF TABLES

1	Address Space Layout Randomization (ASLR) implementation on Windows, macOS and Ubuntu. Legend: ✓= Application Restart, ✓/✗= System Restart, ✗= Never.	21
2	Excerpt of C++ objects in Internet Explorer containing a large number of virtual functions	32

LISTINGS

1	reverse_string() contains multiple memory corruption vulnerabilities.	10
2	Disassembled code that creates the CButtonLayout object	33
3	Disassembly of an indirect call that is instrumented by IFCC. . .	45
4	ZwWaitForSingleObject System Call on Windows 7 32-bit.	45
5	Example IFCC assembly before fix	47
6	Example IFCC assembly after fix	48
7	Example C++program that demonstrates the concept of virtual functions.	52
8	Example C++program that demonstrates virtual functions	55
9	bic masking example	95
10	tst masking example	95
11	Return-address hiding example. Note that constant pool entries are embedded in non-readable memory, as described in Section 4.2.2.2.	99

ACRONYMS

ABI	Application Binary Interface
API	Application Programming Interface
ASLR	Address Space Layout Randomization
BBL	Basic Block
CET	Control-flow Enforcement Technology
CFGuard	Control-flow Guard
CFG	control-flow graph
CFI	Control-flow Integrity
COOP	Counterfeit Object-oriented Programming
COTS	commercial off-the-shelf
CPI	Code Pointer Integrity

CPU	Central Processing Unit
DEP	Data Execution Prevention
DOP	Date-oriented Programming
EPT	Extended Page Tables
IoT	Internet of Things
ISA	Instruction Set Architecture
JIT-ROP	Just-in-Time Return-oriented Programming
JIT	Just-in-Time
JOP	Jump-oriented Programming
LBR	Last Branch Record
plt	Processor Linkage Table
RILC	Return-into-libc
RISC	Reduced Instruction Set Computer
ROP	return-oriented programming
SFI	Software Fault Isolation
SGX	Software Guard Extensions
TB	Tor Browser
TLB	Translation Lookaside Buffer
TRaP	<i>Translation and Protection</i>
vtable	virtual table
XoM	eXecute-only Memory

INTRODUCTION

Our modern society is dominated by computer systems. Nearly every task in our daily life depends on the availability and proper functioning of computer systems in different form factors: desktop, smartphone, and tablets. These systems store, process and transmit security, privacy, and safety critical data. However, the increasing complexity of these systems comes at the cost of an increased attack surface. Attackers exploit security vulnerabilities in software, which executes on these systems, with the ultimate goal to take control of the underlying computing platforms. There are various types of vulnerabilities that range from misconfiguration, e.g., weak passwords, design flaws, or low-level memory-corruption vulnerabilities. Software written in unsafe languages, like C and C++, is particularly vulnerable to the latter type of vulnerabilities because they require manual memory management. Ensuring correct memory management is a highly challenging task, particularly, for software such as operating systems, browsers and document viewers which are comprised of millions of lines of code.

Any mistake while handling memory buffers can lead to a so-called *memory-corruption vulnerability* which allows attackers to access the memory of a vulnerable application in an unintended way. One common mistake during the access of a memory buffer is a missing bounds check. The attacker can exploit the missing check and force the application code to access memory that is beyond the bounds of an allocation memory buffer. As a consequence, the application deviates from its intended behavior. For example, the infamous Heartbleed bug [133], which affected a widely-used SSL library, allows the attacker to read past the bounds of an allocated buffer. Attackers exploited this vulnerability to obtain the private cryptographic keys of affected servers. Another recent buffer over-read vulnerability [161], which affected one of the internal components of the content delivery network provider Cloudflare, resulted in the unintended appending of other users' data to the response of web-requests.

In many cases, memory-corruption vulnerabilities do not limit the attacker to reading memory but enable the attacker to overwrite memory of a vulnerable application during run time as well. Attackers exploit this capability to overwrite code pointers that are then used by the application to set the control flow. As a consequence, the attacker controls which code is executed next. By injecting new code (code-injection attack), the attacker can execute arbitrary malicious payloads within the context of a vulnerable application. Code-injection attacks are conceptually easy to mitigate by enforcing a Writable \oplus Executable ($W\oplus X$) memory policy which ensures that the attacker cannot modify existing code, and cannot execute data [136, 159]. However, attackers adapted their strategies after the widespread adoption of the $W\oplus X$ memory policy. Instead of injecting new code, attackers started to reuse existing code by combining existing code chunks (code-reuse attack) [117, 192]. This attack technique is much harder to mitigate because defenders have to differentiate between a benign and a malicious execution path

within the application code. Code-reuse attacks are an effective attack technique that is used in the real world to completely compromise computer systems that range from mobile phones [63] and browsers on desktop systems [195, 224] to voting machines [34].

The obvious solution to this problem would be the avoidance of unsafe languages in the first place. However, this would require billions of lines of code to be rewritten, a better programming language to be utilized, and adequate training of developers, all of which is unlikely to occur in the near future. Even if the programming language is changed to a more secure one, this is unlikely to completely solve the issue of software vulnerabilities. For example, Java applications do not suffer from memory errors but are prone to bugs of other vulnerability classes that are often easier to exploit [219]. Ironically, the runtime environment is still written in unsafe languages and contains exploitable memory errors [64].

To address the problem of memory-corruption vulnerabilities, industry and academic research groups developed different defense-techniques that aim to mitigate attacks in the presence of memory errors. To this date, the most successful strategies are Stack Cookies [48], Writable \oplus Executable (W \oplus X) memory [136, 159], Address Space Layout Randomization (ASLR) [97, 150, 218] and Control-flow Integrity (CFI) [5, 107, 137, 215]. It is indisputable that research has raised the bar for exploiting memory-corruption vulnerabilities, and for conducting code-reuse attacks. Yet, researchers continue to push the limits of code-reuse attacks and defenses. This arms race has generated many important insights on how and to what extent we can tackle the security threat posed by memory-corruption vulnerabilities. Nevertheless, as we will show, there are still a number of challenges left to mitigate sophisticated code-reuse attacks.

1.1 GOALS AND SCOPE OF THIS DISSERTATION

The main goals of this dissertation are

1. to develop novel attack techniques to bypass state-of-the-art code-reuse attack mitigations, and
2. to introduce the design and implementation of practical leakage-resilient code-randomization schemes to mitigate code-reuse attacks.

The research of run-time defenses against code-reuse attacks can be categorized into randomization-based [119] and control-flow integrity-based [29] defenses. In order to conduct a code-reuse attack the attacker modifies memory addresses of the targeted application during run time by means of a memory-corruption attack. This requires exact knowledge of the memory layout of the target application because any mistake can crash the target, and hence, terminate the attack. Randomization-based defenses aim to mitigate code-reuse attacks by increasing the diversity of the memory layout between two executions of a targeted application. This increases the likelihood that the attacker will make a mistake during the corruption of the memory. Control-flow integrity, on the other hand, verifies the integrity of a subset of memory addresses. Specifically, it checks the integrity of *code pointers* which are used by the application to determine the target

of a branch. Both approaches have been subject to intense research—this dissertation contributes to this research to understand their benefits and limitations—and are being integrated into the real-world software [97, 137, 150, 215, 218].

In this dissertation, we focus on the execution of memory-corruption attacks. Thus, we assume that the attacker already discovered a memory-corruption vulnerability in the targeted application, and a way to take advantage of it. While process of discovering vulnerabilities is related to the topic of this dissertation, it is another line of research [194, 235], and thus, out-of-scope for this dissertation.

1.2 SUMMARY OF CONTRIBUTION

To summarize, the main contributions of this dissertation are as follows:

Memory-disclosure Attacks. We introduce the notion of *direct* and *indirect* memory-disclosure attacks, and demonstrate how adversaries can utilize these techniques to bypass all code-randomization schemes [119] that aim to prevent code-reuse attacks [192]. Direct memory-disclosure attacks exploit the fact that code sections in modern systems are readable, hence, attackers, with the capability to disclose arbitrary memory during run time, can disassemble the randomized code, analyze it, and adjust their code-reuse attack on-the-fly.

At first, it seems such attacks are mitigated by preventing read access to the code section [16, 17]. However, we also show that indirect memory-disclosure attacks, which do not require read access to code section, are as powerful as direct memory-disclosure attacks. Specifically, we demonstrate how attackers can combine offline knowledge of the target binary with run-time information, such as code pointers, to bypass code randomization. Our work on memory-disclosure attacks spawned a new line of research that investigates different techniques to harden code randomization against disclosure attacks [16, 17, 21, 51, 52, 80, 124, 206, 229].

Attacks on Control-flow Integrity. We investigate different Control-flow Integrity (CFI) schemes [4, 137, 215], and present code-reuse attacks that can fully bypass them. Coarse-grained CFI, as it is currently deployed by Microsoft in Windows 10 [137], allows indirect branches to target any valid branch target. We show how the attacker can exploit counterfeit C++ objects to chain multiple C++ virtual function calls together to achieve arbitrary code-execution without violating the integrity checks of coarse-grained CFI.

This attack can be mitigated through compiled-based fine-grained CFI. We take this as a motivation to perform a security analysis of two fine-grained CFI implementations by the most popular open-source compilers, Clang/LLVM and GCC. Our results show that both compilers introduce security vulnerabilities in the conceptually secure fine-grained CFI implementation [215]. Specifically, the applied code optimizations have the unintended side effect that values in registers, which are supposed to be read-only, are temporarily spilled to writable memory. This also affects read-only values, which are used during the enforcement CFI, and hence, gives the attacker a small time window to tamper with these values which results in a full bypass of the CFI enforcement.

Memory-Disclosure Resilient Code Randomization. We present the design and implementation of the first practical and effective mitigation against direct *and* indirect memory-disclosure attacks [61, 196]. For this purpose, we leverage hardware virtualization to implement a primitive that enforces execute-only memory. We then utilize this primitive to prevent direct memory-disclosure attacks by mapping the code section as execute only. Further, we introduce a novel technique, called code-pointer hiding (CPH), which uses execute-only memory as a primitive to mitigate indirect disclosure attacks. In particular, CPH creates an indirection for code pointers that cannot be resolved by attackers who can read and write arbitrary data.

One disadvantage of our technique is that it relies on hardware virtualization support. While almost all modern desktop Central Processing Units (CPUs) and selected mobile CPUs support hardware-accelerated virtualization, this is not true for most embedded CPUs. To tackle this shortcoming, we design and implement a Software Fault Isolation (SFI)-inspired [132, 187] software-based execute-only memory technique for Reduced Instruction Set Computer (RISC)-based CPUs. The impact on the overall run-time performance is low, just as in our virtualization-based approach, however, it does not require *any* hardware-enforced memory protection.

Data-only Attack on JIT compilers. We present a novel data-only attack against Just-in-Time (JIT) compilers that enables arbitrary code execution. Contrary to code-reuse attacks, data-only attacks do not alter the control flow but the data flow of attacked systems. Generally, this limits the expressiveness of data-only attacks. For example, previous data-only attacks disclosed secret information, like cryptographic keys, or escalated privileges [38, 99]. Our data-only attack targets the intermediate representation of the JIT compiler. As a consequence, the JIT compiler generates attacker-controlled native code. Our attack cannot be mitigated through existing defenses like code randomization [119], or control-flow integrity [5, 137, 215], and highlights the power of data-only attacks.

Randomization in the Kernel. We present the design and implementation of a novel mitigation of data-only attacks against page tables. Recently, researchers published the first open-source fine-grained control-flow integrity implementation for the Linux kernel [166]. We perform a security analysis of CFI for the kernel, and our results show that existing CFI implementations for the kernel can be bypassed by manipulating the page tables which are used to define memory access permissions that are enforced by the hardware. Previous work to mitigate attacks against the page tables focused on implementing policy-based access control to the page tables that either introduces high run-time overhead, or depend on certain hardware features [14, 15, 53, 77, 176, 200]. Our work, on the other hand, follows a randomization-based approach resilient to memory-disclosure attacks. In particular, our mitigation ensures that the page tables are placed at a random location, and all references to this location are protected through an indirection that can only be resolved by benign code. Our mitigation comes with the advantage of having no dependencies on special hardware features, and a negligible performance overhead.

1.3 OUTLINE

This dissertation is structured as follows: in Chapter 2 we provide a comprehensive overview on memory-corruption attacks and defenses. Therefore, we first introduce the basics of a memory corruption vulnerability, and then provide a detailed explanation of how attackers exploit these vulnerabilities to compromise a vulnerable system. We follow with a description of countermeasures against code-injection and code-reuse attacks. Specifically, we cover the principals of data execution prevention, address space layout randomization, and control-flow integrity. In Chapter 3 we present three novel memory-corruption attacks against randomization- and control-flow integrity-based code-reuse defenses. Our first attack demonstrates how direct and indirect memory-disclosure attacks are leveraged to bypass code-randomization-based defenses (Section 3.1). Our second attack targets control-flow integrity. We demonstrate how attackers can exploit an imprecise control-flow integrity policy to conduct Turing-complete code-reuse attacks within the policy boundaries. Further, we highlight the importance of binary analysis by showing how the code optimization, when applied to two popular open source compilers, introduce a security vulnerability to an otherwise sound control-flow integrity implementation. We conclude this chapter by demonstrating a novel data-only attack against the intermediate representation of a JavaScript just-in-time compiler. Contrary to previous work, our attack enables the attacker to generate arbitrary malicious code. In Chapter 4 we turn our attention to the mitigation of code-reuse attacks by means of code randomization. Based on our insights from Chapter 3, we design leakage-resilience code randomization. Specifically, we explore the potential of execute-only memory as a primitive to mitigate information-disclosure attacks (Section 4.1). While results show that execute-only memory is a suitable primitive, execute-only memory is not natively supported through paging. Our initial prototype relies on a hypervisor to enable execute-only memory for x86 desktop systems, however, most embedded systems do not feature hardware virtualization. We overcome this limitation by implementing software-based execute-only memory (Section 4.2). We further present the design of a linker wrapper to create self-randomizing binaries (Section 4.3). Finally, we present a novel randomization scheme to protect kernel page tables against data-only attacks (Section 4.4). We conclude this dissertation in Chapter 5

1.4 PREVIOUS PUBLICATIONS

This dissertation is based on the following peer-reviewed publications. The full list of publications published by the author of this dissertation can be found in Chapter 6.

Chapter 3: Advances in Memory-Corruption Attacks

1. Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of

- Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.
2. Luca Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, Fabian Monroe. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2015.
 3. Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
 4. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.
 5. Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, Michael Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
 6. Tommaso Frassetto, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. JITGuard: Hardening Just-in-time Compilers with SGX. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

Chapter 4: Advances in Memory-Corruption Defenses

1. Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.
2. Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Ahmad-Reza Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2016.
3. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi. Selfrando: Securing the Tor Browser against De-anonymization Exploits. In *Proceedings of the Annual Privacy Enhancing Technologies Symposium (PETS)*, 2016.

4. Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

BACKGROUND

In this chapter, we provide the background on memory-corruption attacks, that are typically enabled by implementation errors while using low-level languages, such as C and C++. These languages allow for flexible and efficient programming, and hence, are used for almost all modern software. One of their properties is allowing unrestricted access to the memory. This comes with the disadvantage of offloading the responsibility of ensuring that all memory accesses are safe onto the programmer. In the best case, failing to ensure safe memory accesses leads to a crash of the application. In the worst case, it enables the attacker to completely compromise the application by providing an input to the targeted application that results in the access of an unintended section of the application's memory. From here on we will call an unsafe or unintended memory access, which leads to an attacker-controlled deviation of the programmer's intended behavior of the application, a *memory-corruption vulnerability*, and its exploitation *memory-corruption attack*.

In the following pages, we provide the necessary concepts and technical background which are required to understand the remainder of this dissertation. Therefore, we first introduce memory-corruption vulnerabilities (Section 2.2), and techniques used by attackers to exploit these vulnerabilities to take complete control of the targeted application. Next, we give an overview how memory-corruption attacks and defenses evolved over time. This evolution can be roughly categorized into code-injection attacks and defenses (Section 2.3), and code-reuse attacks and defenses (Section 2.4).

In general, memory-corruption attacks and defenses are tailored to the underlying Central Processing Unit (CPU) architecture of the targeted system, the concepts are applicable across different CPU architectures. However, most of the attacks and defenses, which we present in this dissertation, target x86-based desktop system, hence, we use x86 assembly instructions in this section for our explanation if required.

2.1 LOW-LEVEL VIEW OF AN APPLICATION

Listing 1 contains the source code of a function of a vulnerable application which we will use as an example throughout this section. It reverses a string by first copying the input string to a temporary buffer (line 5), and then overwriting the input string in reverse with the content of the temporary buffer (line 7-9). `reverse_string()` contains two vulnerabilities: the first vulnerability is a buffer overflow (line 5) due to the use of `strcpy()`, and the second vulnerability is an information leakage (line 8) due to a missing length check.

Exploiting both vulnerabilities requires knowledge about the low-level view of an application. Hence, we will first discuss the general layout of an application during run

```

1 void reverse_string(unsigned char *buf, unsigned int buf_len) {
2     unsigned char tmp_buf[64];
3     int i;
4
5     strcpy(tmp_buf, buf);
6
7     for(i = 0; i < buf_len; ++i) {
8         buf[i] = tmp_buf[buf_len - 1 - i];
9     }
10 }

```

Listing 1: `reverse_string()` contains multiple memory corruption vulnerabilities.

Main Application			Shared Libraries			Heap	Stack	Kernel
RX	RX	RWX	RX	RX	RWX	RWX	RWX	
Code	Data	...	Code	Data	...	Data	Data	Code/ Data

Figure 1: High-level memory layout and access permissions of an application during run time.

time before explaining how the attacker can exploit these vulnerabilities to take full control of the application.

Application Memory Layout

Figure 1 contains a simplified view of the virtual memory layout of modern applications during run time. The memory region of the main application and (multiple) shared libraries is generally divided into a code and data section. On x86 *paging* is used to enforce memory-access permission. Unfortunately, paging does not allow setting the read-write-execute permissions individually for a memory region. Instead, the permissions for a memory region can be set to one of three options: non-accessible; to readable and executable; or to readable, writable and executable. The data sections are considered to be statically-allocated memory, and commonly consist of read-only data (constant variables), and writable data (global variables). For dynamic allocations two separate memory regions are used: the heap for global memory allocations, and the stack for local, i.e., function-call-specific, memory allocations. Generally, memory regions are separated by non-accessible memory. Finally, the operating system’s kernel is mapped into the process space but not accessible to the application.

We now provide a brief description of the data structures that are used to organize the stack as the vulnerabilities in Listing 1 are stack-related. We note that such vulnerabilities can also affect heap-allocated memory, and that the concepts we discuss with the help of our stack-related example can be applied to heap-based vulnerabilities.

Stack Memory Organization

On x86 the stack is a memory region that grows towards to lower addresses, and is divided into *stack frames*. A stack frame is an Application Binary Interface

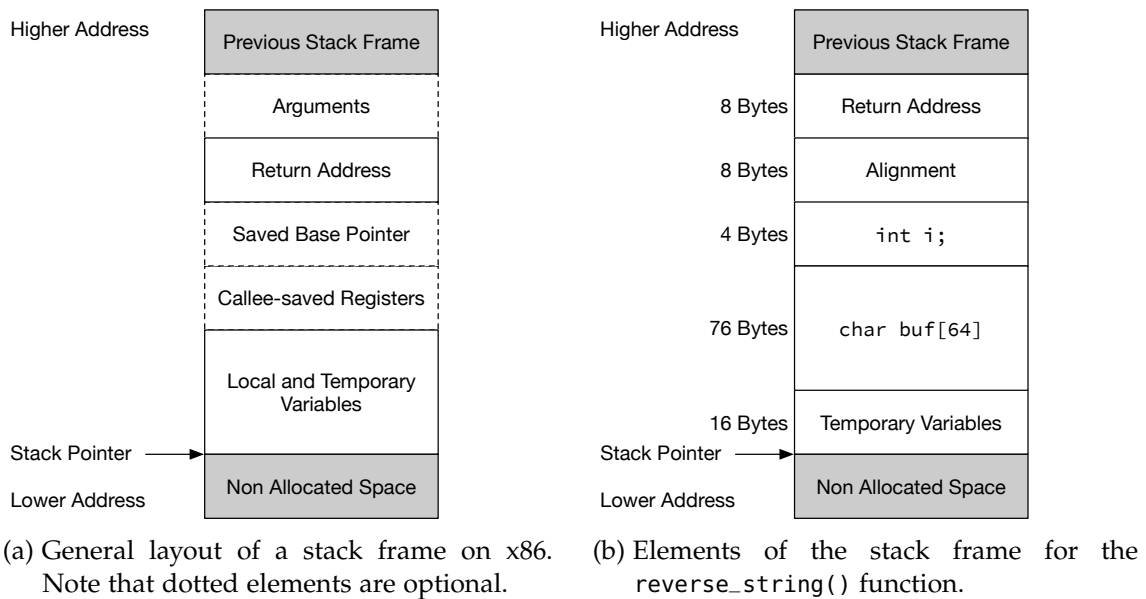


Figure 2: Stack frames on x86.

(ABI)-dependent data structure, which the application allocates during a function call, and releases when the function returns. Figure 2a illustrates the general layout of a stack frame on the x86 architecture. Depending on the calling convention, the first few arguments are written onto the stack (common on x86 32-bit) or are passed through registers (common on x86 64-bit). Independently from the calling convention and architecture, the next element on the stack is the return address. The return address is a code pointer, which is automatically written by the `call` instruction and points to the next instruction after the `call` instruction. If not disabled through a compiler flag, functions use the base pointer register to address local variables by pointing it to the current stack frame, however, before overwriting the base register, functions save the current base pointer value on the stack. Similarly, if a function uses callee-saved registers for its computation, then the current values of these registers are temporarily stored on the stack as well. Finally, each function allocates space for local variables and temporary values, and the stack pointer is set to the beginning of the current stack frame.

Figure 2b shows the stack frame of the `reverse_string()` function. The actual layout of a stack frame does not only depend on the Central Processing Unit (CPU) architecture and ABI but also on the compiler. It can allocate additional space to align memory addresses for faster memory access, or to optimize the code of a function such that a local variable can be stored in a register instead of on the stack. For example, instead of 64 bytes as declared by the programmer, the compiler generates code that reserves 76 bytes for the buffer, and adds another eight bytes between the counter variable `i` and the return address.

Hence, when the attacker provides an input that is larger than 76 bytes the `strcpy()` function will overflow the buffer and overwrite other variables, or even the return address.

Next, we shortly explain what a control-flow graph (CFG) is and conclude our brief introduction of the low-level view of an application.

Control-flow Graph

A CFG is a graph representation of all benign execution paths of an application during run time. Each vertex in the CFG represents a Basic Block (BBL), and each edge represents a valid execution path from one BBL to another. A BBL consists of a number of assembly instructions, and has exactly one entry point and one exit point which is a branch instruction. In general, we distinguish between *direct* and *indirect* branches. A direct branch encodes the destination address of the branch within the instruction, whereas an indirect branch encodes the data-memory address or register within the instruction that contains the destination address. For example, the aforementioned return address, which is written by a call instruction, and then used by a return instruction, is such an indirect branch instruction.

2.2 MEMORY-CORRUPTION ATTACKS

In general, memory-corruption vulnerabilities are categorized into *spatial* corruption, where the application accesses a memory buffer outside of its bounds, or *temporal* corruption, where the application accesses memory before its initialization or after it was released. An example of a spatial corruption is the classic buffer-overflow vulnerability where a missing bound check leads to the corruption of adjacent memory. Examples of temporal corruptions are uninitialized memory, and use-after-free vulnerabilities. In the former case the application reads a memory value without proper initialization or verification. If the attacker can set this memory value before it is read, the application performs its computation based on a bogus value which, depending on what the value is used for, either leads to further memory corruptions, or control of the application. The latter describes the case where the application holds more than one reference to an allocated memory buffer, which is then released without invalidating all references. Use-after-free vulnerabilities are particularly common in C++ applications which follow a modular design, like modern browsers, document viewers, and office applications.

In the following section, we discuss memory-corruption vulnerabilities based on a buffer overflow vulnerability in detail, and how attackers can exploit such vulnerabilities to hijack the control flow, or disclose arbitrary memory.

Buffer Overflow

In the past, Application Programming Interface (API) functions, which manipulate buffers without checking the bounds of the source or destination buffer, were the main cause for buffer overflows. The most prominent example is the `strcpy(dst, src)` function which copies a string from the source buffer into the destination buffer. However, the `strcpy()` function does not verify that the source buffer fits into the

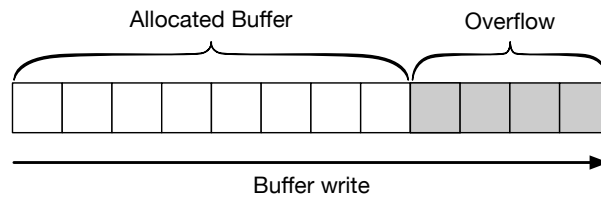


Figure 3: During a buffer overflow the attacker writes past the allocated buffer bounds.

destination buffer. As a consequence, the attacker can exploit `strcpy()` to overwrite memory adjacent to the destination buffer by providing a source buffer that is larger than the destination buffer.

Figure 3 shows an example of a buffer overflow. Here, the application allocates a buffer of eight slots. However, the write operation writes twelve slots to the buffer. As mentioned before, C and C++ do not check the bounds of a buffer during a write operation, hence, the write operation will overwrite (*overflow*) the adjacent memory which can be in use by the application to store other variables or temporary values.

Control-flow hijacking

The attacker can exploit the buffer-overflow vulnerability in the `reverse_string()` (Listing 1, line 5) and the knowledge of the stack layout for this function (Figure 2b) to overwrite the return address of the current stack frame. As a consequence, the return instruction uses an attacker-controlled value as a destination address. This enables the attacker to add a new (malicious) edge to the control-flow graph (CFG), and force the application to behave differently from the programmer's intention. To fully take control of the targeted application, attackers execute either *code-injection* or *code-reuse attacks*.

Before we describe these attack techniques in detail, we explain how the second vulnerability can be exploited to disclose memory.

Memory disclosure

The second vulnerability in `reverse_string()` (Listing 1) is due to an unchecked length value. Specifically, to avoid this vulnerability the function should have verified that the length passed as an argument matches the length of the input string. Hence, by providing a length value that is larger than 64, the attacker can trick the application into writing the content adjacent to the allocated temporary buffer to the output buffer. This enables the attacker to disclose secrets, e.g., memory addresses, stored on the stack.

2.3 CODE-INJECTION ATTACKS AND DEFENSES

In the previous section, we discussed how the attacker can exploit a memory-corruption vulnerability to change the control flow to an attacker-controlled address. To execute malicious code in the context of the vulnerable application, the attacker can perform a code-injection attack. The high-level idea is that the attacker injects new vertices into the control-flow graph (CFG) of the application, and then creates an edge to the injected Basic Block (BBL) as shown in Figure 4.

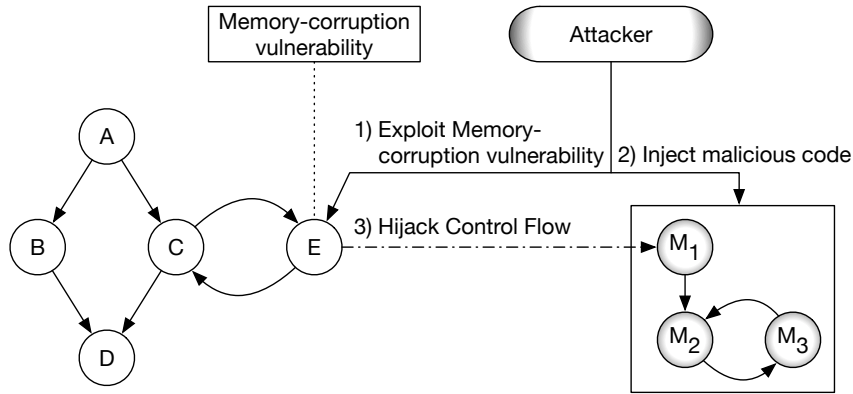


Figure 4: Code-injection Attack

Code-injection attacks are possible due to the permission system of paging on the x86 architecture (cf. Section 2.1), which does not distinguish between the read permission and the execute permission. Hence, the attacker can write a malicious program into a data buffer, and overwrite a code pointer, which is subsequently used as a branch target by the application, to point to the data buffer [8]. In the past, attackers leveraged code-injection attacks to obtain a remote shell. Therefore, the injected malicious code is often referred to as *shellcode*.

Figure 5 shows a code-injection attack against our vulnerable example application of Listing 1. The attacker provides a buffer that contains first the shellcode, then some padding that fills the rest of the buffer, and a code pointer which overwrites the return address and points to the beginning of the injected shellcode. For a successful attack, it is important that the attacker knows the exact address of the shellcode on the stack. This is provided by the fixed memory layout of applications during run time, i.e., all memory regions as shown in Figure 1 (Section 2.1) are loaded to the same address.

Writable xor Executable Memory Policy

Code-injection attacks are mainly enabled by the fact that the x86 architecture does not distinguish between code and data. The reason for this is that the x86 architecture follows the von Neumann architecture that allows mixing code and data in memory—as opposed to the Harvard architecture, which requires separate memory for code and data. However, with the exception of programs that generate code during run time, regular programs do not require data memory to be executable. Hence, removing the executable

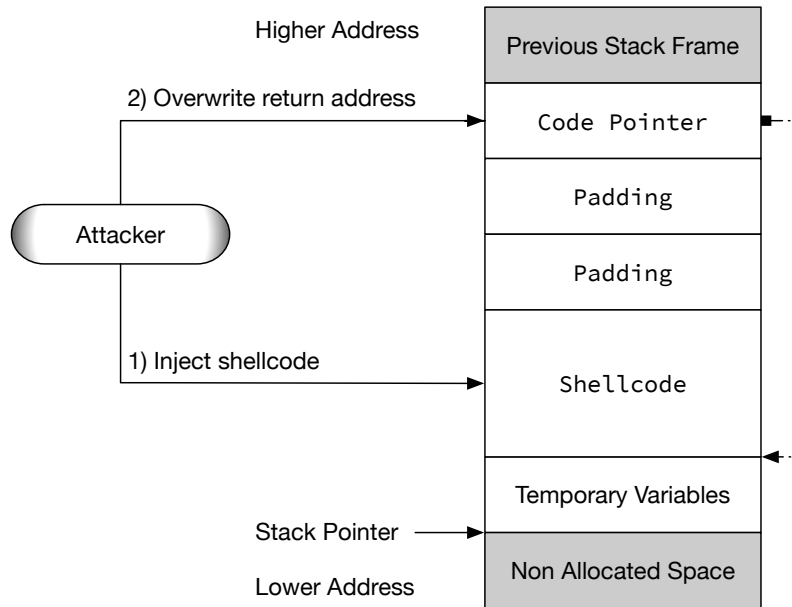


Figure 5: Stack layout during a code-injection attack.

permission from data memory regions is an effective way to generically prevent code-injection attacks. This memory protection policy is known as Writable \oplus Executable ($W\oplus X$) because memory pages can be either executable or writable but not both at the same time.

To overcome the limitations of the x86 paging permission system, previous research leveraged segmentation [208], which is a legacy memory protection mechanism present on x86 Central Processing Units (CPUs). Alternatively, a technique referred to as Translation Lookaside Buffer (TLB) splitting [207] exploits that instruction and data reads use different caches to store paging information which also contain the memory access permissions. Both approaches come with disadvantages: segmentation only allows the enforcement of memory access permission for larger segments, hence, the application space must be split into fixed-size segments during the application start, while TLB splitting comes with a non-negligible performance overhead. Fortunately, paging on modern CPUs was extended to include the *non-executable* memory permission. This allows modern operating systems to mark all data memory as non-executable, and therefore, to prevent code-injection attacks [136].

Next, we discuss how attackers adapt their strategy to overcome the challenge of $W\oplus X$ memory.

2.4 CODE-REUSE ATTACKS AND DEFENSES

The enforcement of a $\text{Writable} \oplus \text{Executable}$ memory policy mitigates code-injection attacks. Therefore, attackers adapted their strategy from injecting malicious code to chaining existing code to perform the same malicious action as the injected code. Figure 6 shows the general idea of code-reuse attack. Similar to code-injection attacks, the attacker first exploits a memory-corruption vulnerability. However, instead of injecting malicious code, the attacker injects malicious data in the form of code pointers. Generally, the chaining of existing code requires that the reused code ends in an indirect branch instruction.

Now we discuss the two most prominent code-reuse attack techniques, *return into libc* and *return-oriented programming*, and then two effective mitigation techniques: *code randomization* and *control-flow integrity*.

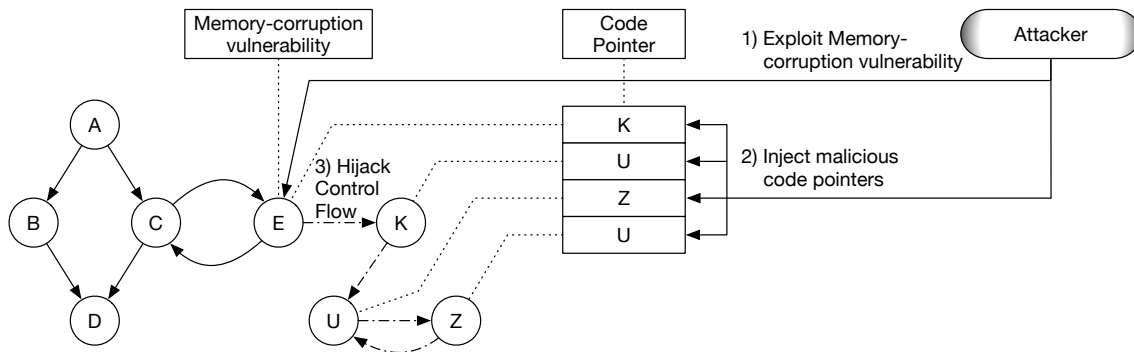


Figure 6: Code-reuse Attack

Return into libc

Solar Designer [198] was the first to provide a practical instantiation of a code-reuse attack technique to bypass $W \oplus X$ memory, called Return-into-libc (RILC). He exploited a memory-corruption vulnerability to overwrite the return address with a pointer to a function of libc. libc is the standard C library and, aside from basic functions to process strings and numbers, it also provides wrapper functions to invoke system calls. System calls are the Application Programming Interfaces (APIs) to the kernel functions to, e.g., read files from the hard disk, communicate over the network, or start new processes. The main idea of RILC attacks is to change the control flow of an attacked application consecutively such that it invokes a number of libc functions with attacker-controlled arguments to achieve a specific behavior.

In the past, a common code-injection attack payload for Linux was to inject code that would execute the system call `execve()`, to execute a shell, like `/bin/sh/`, with the privileges of the vulnerable application. Solar Designer's RILC attack achieved the same by exploiting a stack-based buffer overflow to overwrite the return address to point to the `system()` function of the libc. `system()` takes the path of an application as an argument and eventually calls the `execve()` system call. His exploit targeted a

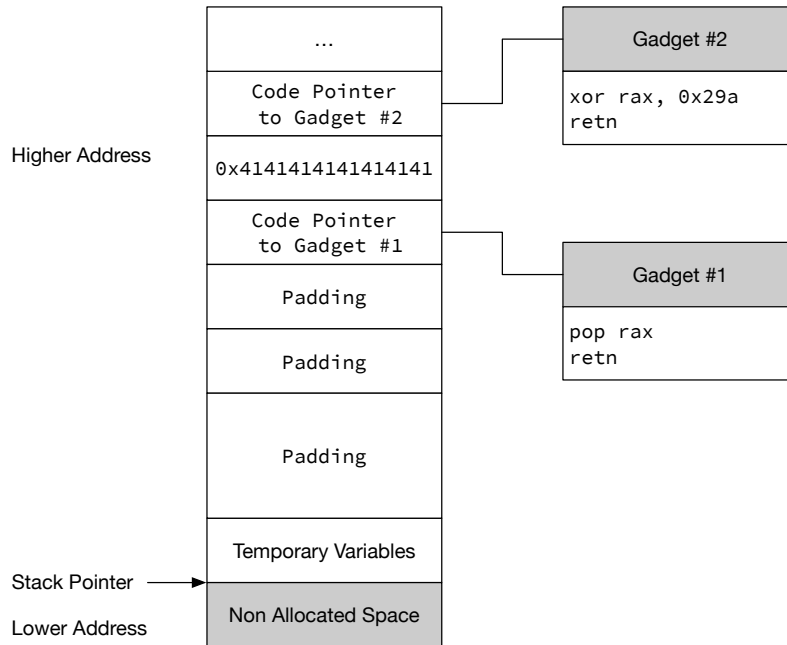


Figure 7: Return-oriented programming attack

vulnerable application on the x86 32-bit architecture. As we mentioned in Section 2.1, on x86 32-bit function arguments are passed through the stack. Hence, if the attacker exploits a stack-based buffer overflow she can control the arguments passed to a function. However, on x86 64-bit and other popular architectures, like ARM, arguments are passed through the register, hence, the original exploitation technique from Solar Designer would not work.

Apart from being incompatible with those architectures, RILC suffers from other weaknesses as well. Most importantly, the attacker is limited to the functionality implemented by loaded shared libraries. While in practice it is unlikely that such imposed constraints would stop an attack, it increases the difficulty of creating attack payloads. For example, an attack payload might require one to perform pointer arithmetic for a successful execution. Another weakness of RILC is that even if arguments can be passed through the stack, those arguments might need to include NULL bytes. The problem arises if the cause of the buffer overflow is a string manipulation function, like `strcpy()` in Listing 1. `strcpy()` copies byte-wise content of the source buffer into the destination buffer until the source buffer reaches a NULL byte which indicates the end of the input string. Hence, an RILC attack payload for this vulnerability cannot include NULL bytes in the middle of the payload, otherwise `strcpy()` will truncate the payload during the overflow.

Next, we discuss *return-oriented programming* [192], which is a generalization of *borrowed code chunks* [117]. return-oriented programming (ROP) reuses small instruction sequences instead of whole functions to overcome the limitations of RILC.

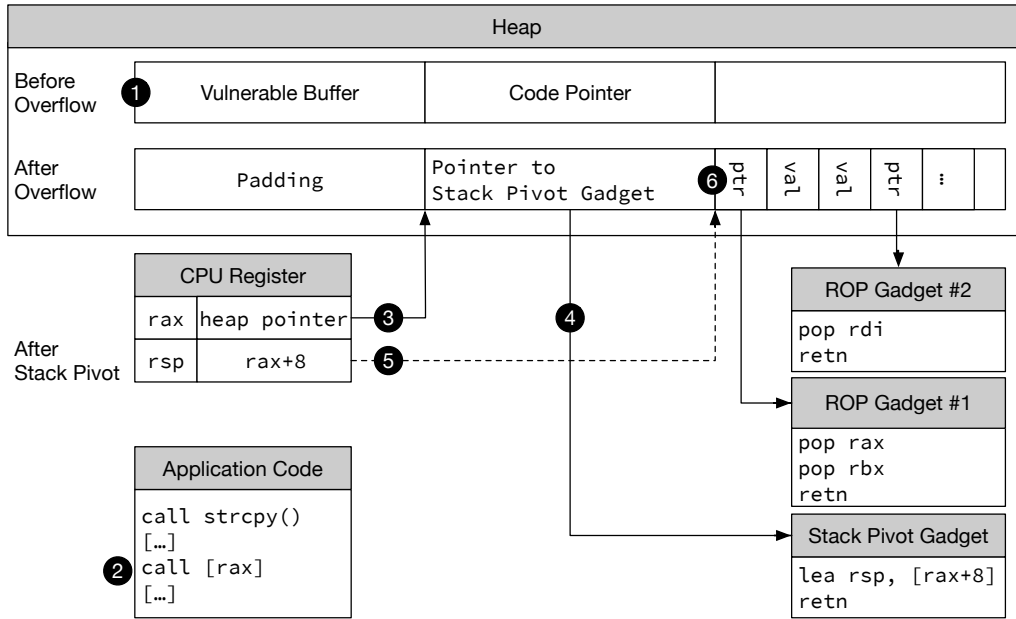


Figure 8: Attacker exploits a heap-based buffer overflow and leverages a stack pivot gadget to launch a ROP attack.

Return-oriented Programming

Krahmer [117] was the first to present a code-reuse attack technique that was later generalized by Shacham [192] to what is known as ROP. The main idea of ROP is to chain short instruction sequences that end with a return instruction instead of whole functions. Each short instruction sequence performs a specific task, e.g., loading a value into a register, or adding the values of two registers. Shacham [192] defines a *gadget* as one or more short instruction sequences that, when combined, perform a high-level task like reading a value from a memory address and writing it to another. Further, he showed that ROP is Turing-complete which means that, contrary to RILC, ROP attacks are not limited to the existing code.

Figure 7 shows how to leverage ROP to get code execution for our example application which we use throughout this section. Similar to the code-injection attack the attacker overflows the local buffer, instead of injecting shellcode she overwrites the return address with a code pointer that points to the first gadget. Hence, when the function executes the return instruction it redirects the control flow to the first gadget. Note that the return instruction increases the stack pointer by four (32-bit) or eight (64-bit). Hence, the stack pointer will point to the value `0x4141414141414141` on the stack. The application now executes the two instructions of the first gadget. The `pop` instruction loads the value, to which the stack pointer points to, into the `rax` register, and, increases the stack pointer by eight. Then, the return instruction reads the next code pointer from the stack and changes the control flow to the second gadget. Using this technique, the attacker can chain an arbitrary number of instruction sequences together to execute a malicious payload.

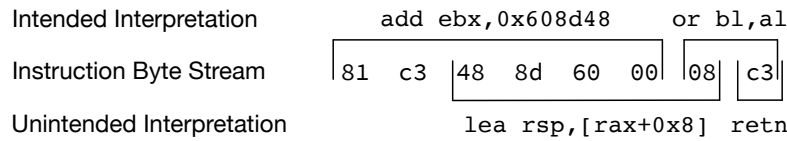


Figure 9: Example of unaligned instructions on x86 64-bit.

In practice, however, there are two common challenges that increase the difficulty of conducting ROP attacks: first, stack-based buffer overflows are less common due to improved compiler mitigations. For example, stack canaries [48] are random values which the compiler places between the local variables and the return address, and whose integrity is verified before the return instruction is executed. Hence, most (exploitable) memory-corruption vulnerabilities are heap buffers. This increases the difficulty of ROP attacks because they require the attacker to inject return addresses on to the stack. Second, similar to RILC attacks, the expressiveness of ROP attacks solely depends on the available gadgets, and in some cases the available code might not contain a gadget which is necessary for a successful attack. Next, we discuss two techniques, called *stack pivoting* and *unaligned gadgets*, that attackers use to overcome both challenges.

Stack pivoting is a technique that enables the attacker to change the stack pointer register to point to an attacker-controlled memory buffer, usually on the heap. Figure 8 shows in detail how this technique works. For our example we assume that the attacker can overflow a buffer on the heap which is followed by a code pointer ❶. The attacker exploits the memory-corruption vulnerability to overwrite the code pointer with a pointer to the stack pivot gadget followed by a regular ROP payload. When the application now performs an indirect call using the register ❷, which points to the heap ❸, the call instruction changes the control flow to the stack pivot gadget ❹. In this case the `lea` (load effective address) instruction overwrites the stack pointer register (`rsp`) with the value of the `rax` register plus eight. This sets the stack pointer to the heap, specifically, to the attacker’s injected ROP payload ❺. Hence, the return instruction of the stack pivot gadget will read its return address from the heap, instead of from the original stack ❻. From here on, the attacker can conduct a normal ROP attack as described above. One practical challenge of this technique is for the attacker to find a suitable stack pivot gadget because overwriting the stack pointer is not a common functionality for benign application code. To increase the odds of finding such a gadget, attackers exploit the fact that x86 architecture allows unaligned access to instructions which enables the attacker to use *unaligned gadgets*.

Unaligned gadgets are instruction sequences that are not generated by the compiler. They exist because the x86 architecture uses variable lengths for its instructions, and hence, each byte of a benign instruction can be interpreted as the beginning of a new—*unaligned*—instruction. Figure 9 shows an example of an unaligned gadget. The compiler generated an `add` and an `or` instruction. However, by jumping into the middle of the benign `add` instruction the Central Processing Unit (CPU) will interpret the

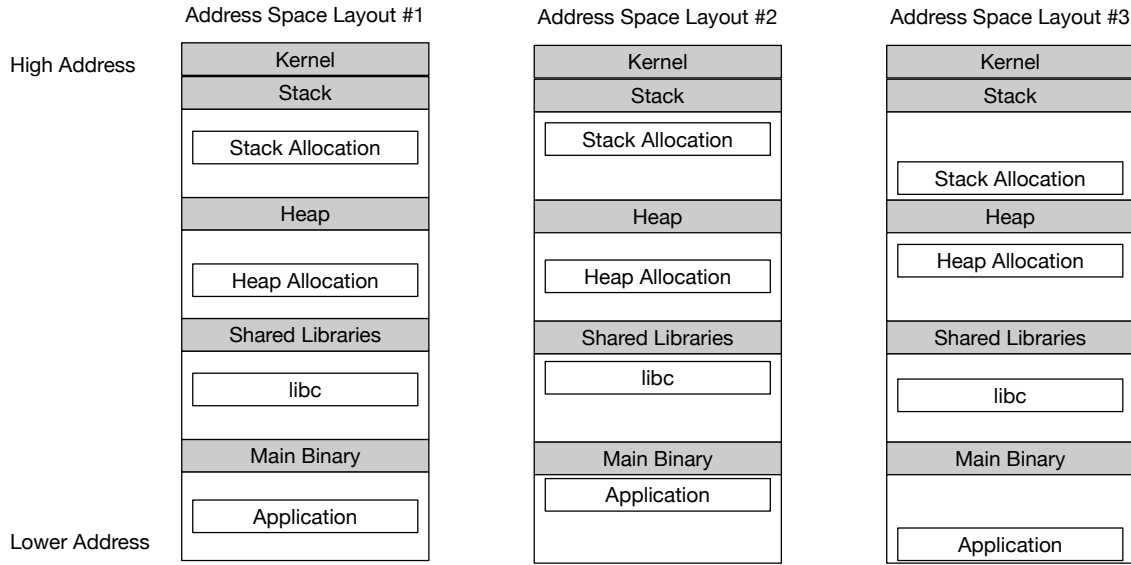


Figure 10: Address Space Layout Randomization changes the base address of code and data sections.

generated instructions as a stack-pivot gadget. Shacham [192] shows that attackers can generate Turing-complete ROP payloads using only unintended instruction sequences.

Code and Data Randomization

A fundamental assumption for launching control-flow hijacking attacks is that the attacker knows the process layout of the targeted application. For example, to conduct a code-injection attack, the attacker must know the address of the injected payload, and for a code-reuse attack she needs the address of each gadget. Randomization-based defenses break the assumption of a deterministic process layout by randomizing the addresses of code and data sections. Randomization can be applied in different granularity levels: the more fine-grained the code and data is randomized the harder it is to guess the randomization offset. However, increased randomization granularity can also negatively impact performance [119].

Most modern operating systems, like Linux [218], Windows [97], and macOS [150], deploy Address Space Layout Randomization (ASLR), which is a coarse-grained randomization technique. ASLR randomizes the base address of the main binary, shared libraries, and dynamically-allocated memory (heap and stack). In particular, the operating systems divide the address space into segments in which they randomly choose a base address, as shown in Figure 10. This design has two major disadvantages. First, it limits the randomization entropy by defining fixed-size segments. This particularly affects systems with a small address space like 32-bit based systems where the randomization offset is brute-forceable [193]. Second, the offsets within a randomized region, for example the relative gadget addresses within the `libc`, or memory allocations, remain unchanged. Hence, it is sufficient for the attacker to disclose

one valid pointer of a randomized segment, e.g., by means of a brute-force attack or a memory-disclosure attack, to know every other address of this segment.

	Windows	macOS 10.12.3	Ubuntu 16.04.01
Memory Region	Visual Studio 2015	clang 8.0	gcc 5.4
Main Binary	✓/✗	✓	✗
Shared Library	✓/✗	✓/✗	✓
Stack	✓	✓	✓
Heap (small)	✓	✓	✓
Heap (big)	✓	✓	✓

Table 1: ASLR implementation on Windows, macOS and Ubuntu. Legend: ✓= Application Restart, ✓/✗= System Restart, ✗= Never.

In practice, the implementation of ASLR varies between different operating systems. We conducted an experiment in which we compiled and executed a program, which records the addresses for the main binary, shared libraries, stack, and small (10 bytes) and big (64 kilobytes) heap allocation on modern 64-bit operating systems. We distinguish between small and big heap memory allocation because heap allocators adjust their allocation strategy based upon the requested memory size. Table 1 summarizes the results. In general, Windows 10 provides the highest entropy for data allocations, but only re-randomizes the location of executable code during system restart. Ubuntu (Linux), on the other hand, does not randomize the location of the main binary by default.

ASLR is the first practical defense deployed in all major operating systems. This forces attackers to leverage memory-disclosure attacks to leak the randomization secret [190] before they can conduct a code-reuse attack. Next, we discuss an alternative, policy-based mitigation technique.

Control-flow Integrity

In general, the control-flow graph (CFG) of applications does not change during run time. An exception to this is applications such as browsers, which include a scripting engine that generates new code during run time [87, 139, 145]. However, during a code-reuse attack the attacker changes the CFG by injecting new edges. Hence, to effectively mitigate such attacks it is sufficient to prevent changes to CFG.

Control-flow Integrity (CFI) [5, 6] is a technique that prevents the attacker from adding new edges to the CFG. In particular, CFI instruments all vertices that exit through an indirect branch instruction with a check that verifies that the statically-computed CFG contains an edge which connects the current and the target branch, as shown in

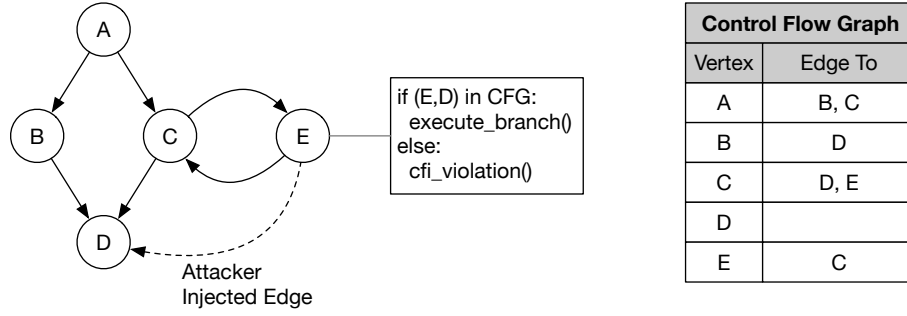


Figure 11: Control-flow Integrity verifies the target of indirect branches before executing them.

Figure 11. If the CFG contains the edge then the branch is executed, otherwise the instrumentation generates a CFI exception.

We distinguish between *forward-edge* (indirect call and jump instructions) and *backward-edge* (return instructions) CFI because their enforcement requires different techniques, which we will discuss later in this section. Further, CFI requires $W \oplus X$ to be in place to prevent the attacker from modifying existing or adding new vertices to the CFG.

Abadi et al. [5, 6] are the first to present a CFI in practice. Their approach leverages static binary analysis and binary instrumentation to harden commercial off-the-shelf (COTS) binaries against code-reuse attacks. In particular, they first generate a CFG by identifying all valid branch targets for each indirect branch. Next, they group the branch targets of each indirect branch and generate a unique id (label) for each target group. Each indirect branch is then instrumented to verify that the target is marked with the label of the corresponding branch target group. In order to label Basic Blocks (BBLs) they insert the x86 instruction *prefetch* at the entry of each BBL. The *prefetch* instruction takes an address as an argument, and hints to the CPU that the application will access this data address in the future. The advantage of this instruction is that it has no impact on the execution state of the application, and does not create an exception if the address is invalid. Hence, its argument can be used to encode unique identifiers (labels).

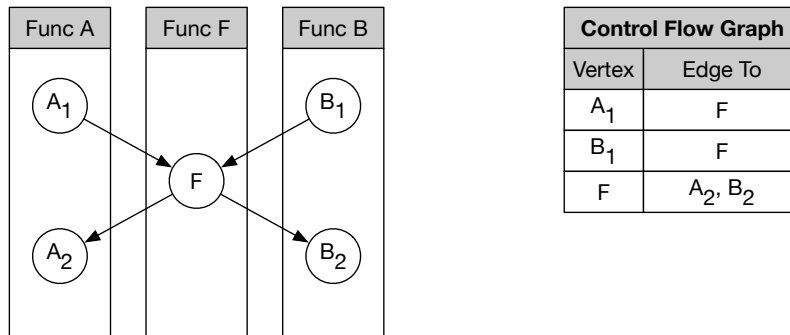


Figure 12: Static verification of return targets is too imprecise.

This static approach comes with the disadvantage that it is imprecise for return instructions. For example, Figure 12 illustrates the part of the CFG where a function

vertex (F) is called by two other vertices. In this example, A_1 branches to F through a `call` instruction. From a semantic point of view, the only valid branch target for the `ret` instruction is A_2 . However, according to the static analysis A_2 and B_2 are possible branch targets, because F can be called by B_1 as well. Hence, the attacker could change the semantics of the application by overwriting the return address that points to A_2 to instead point to B_2 . Carlini et al. [32] demonstrated that a static approach for return instructions gives the attacker enough leeway to conduct arbitrary ROP attacks.

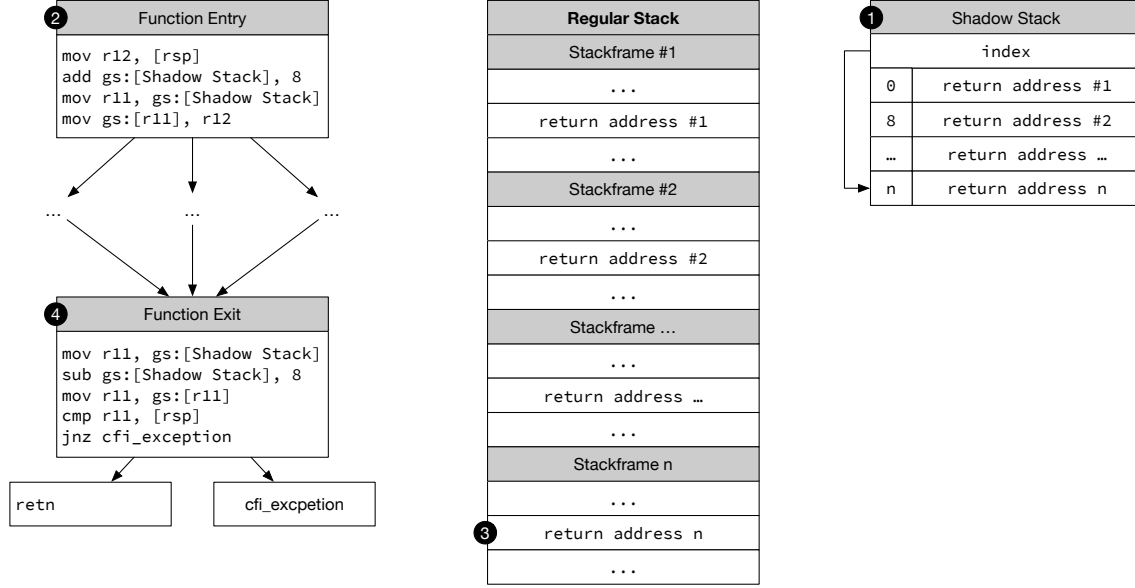


Figure 13: CFI can leverage a shadow stack to enforce that return instructions only return to the call site that invoked the current function.

To increase the precision of CFI for return instructions, Abadi et al. [5, 6] proposed a *shadow stack*, which is a separate, isolated stack that is used to store and verify return addresses. Shadow stacks are highly effective because they enforce the natural semantics of the call/return instructions, i.e., the return instruction is supposed to transfer the control flow to the next instruction after the call instruction that invoked the current function. However, there are benign cases in which an application breaks this semantic that must be considered when enforcing CFI for backward edges. For example, if the currently executed function generates an exception, it is not always the case that the caller function catches the exception. Instead another function in the call hierarchy or the default exception handler can be responsible for handling such an exception. In this case the current function returns to the function in the call stack which implements the exception handler.

We illustrate the functionality of a shadow stack in Figure 13. The shadow stack itself is a traditional stack without a dedicated stack pointer (1). Hence, the first entry is the index to the last added return address. The function entry (2) is instrumented such that a called function first reads the return address from the regular stack (3), and saves it temporarily in a register. Then it increases the index of the shadow stack and saves the return address on the shadow stack. The function exit (4) is instrumented as well.

Specifically, it reads the current return address from the shadow stack, decreases the index, and then compares it to the current return address on regular stack. If both return addresses are the same, the function returns normally, if they are different, a CFI exception is generated. To provide effective isolation, Abadi et al. [5, 6] use x86’s segmentation feature. For architectures that do not provide segmentation as a means to isolate memory, Software Fault Isolation (SFI) [132, 187] can be leveraged.

Validating all indirect control-flow transfers can have a substantial performance impact that prevents widespread deployment. For instance, when validating forward and backward edges, the average run time overhead is 21% for the initially proposed CFI [3] and 13.7% for state-of-the-art solutions (4.0% for forward [215] and 9.7% for backward edges [56]). Several CFI frameworks attempt to reduce the run-time overhead by enforcing coarse-grained policies. There is no clear definition in the literature with respect to the terms *fine* and *coarse-grained* CFI policy. However, the general understanding of a *fine-grained* CFI policy is that only branches intended by the programmer are allowed. In contrast, a *coarse-grained* CFI policy is more relaxed and might allow indirect branches to target the start address of any function. For instance, ROPecker [43] and kBouncer [163] leverage the branch history table of modern x86 processors to perform a CFI check on a short history of executed branches. Zhang and Sekar [239] and Zhang et al. [237] applied coarse-grained CFI policies using binary rewriting to protect COTS binaries. Relaxing the CFI policies (or introducing imprecision to the CFG) has the downside of enabling the attacker to launch code-reuse attacks within the enforced CFG. Consequently, coarse-grained variants of CFI have been repeatedly bypassed [31, 60, 84, 185].

ADVANCES IN MEMORY-CORRUPTION ATTACKS

In this chapter, we present novel memory-corruption attacks against code-randomization- and control-flow integrity-based code-reuse attack mitigations. Specifically, we present two novel information-disclosure attacks in Section 3.1.2 and Section 3.1.3, which target code-randomization schemes. Our attacks highlight the necessity to implement some form of leakage resilience for randomization-based defenses and serve as a motivation for the next chapter. Next, we present three attacks against Control-flow Integrity (CFI) which emphasize pitfalls that need to be addressed during the design and implementation of CFI schemes: our first attack bypasses fine-grained control-flow integrity by exploiting a bug that is introduced during the optimization step of compilers (Section 3.2), our second attack bypasses coarse-grained control-flow integrity by chaining virtual function together (Section 3.2), and our third attack bypasses control-flow integrity by manipulating the intermediate representation of Just-in-Time (JIT)-compilers (Section 3.4). In Section 3.5 we elaborate upon related work on memory-corruption attacks and conclude in Section 3.6.

3.1 MEMORY-DISCLOSURE ATTACKS

Code randomization mitigates code-reuse attacks by preventing the attacker from knowing the exact address of the gadgets that are required for conducting the attack. However, code randomization is vulnerable to memory-disclosure attacks. As shown by Serna [190], Address Space Layout Randomization (ASLR) can be bypassed by disclosing a single code pointer during run time. This is due the fact that ASLR shifts each code segment by a random offset. In order to address this issue, researchers focused on increasing the granularity of code randomization from shuffling the function order in memory, going as far as the randomizing of single instructions. For a comprehensive overview of different code-randomization schemes, we refer to Larsen et al. [119].

However, code randomization remains vulnerable to memory-disclosure attacks independent of its granularity. We make a distinction between *direct* and *indirect* disclosure attacks. Figure 14 shows the high-level idea of both techniques: the main difference is that in a direct memory-disclosure attack the attacker reads the code pages directly to find the required gadgets. In an indirect memory-disclosure attack the attacker reads only code pointers from the data region, e.g., stack or heap, to infer the gadget’s address by combining run time knowledge with an offline analysis of the targeted binary and the applied randomization scheme.

In the following pages we explain in detail how the attacker can exploit direct (Section 3.1.2) and indirect (Section 3.1.3) memory-disclosure attacks to bypass any code-randomization scheme. First, however, we introduce the threat model that we assume for both attacks.

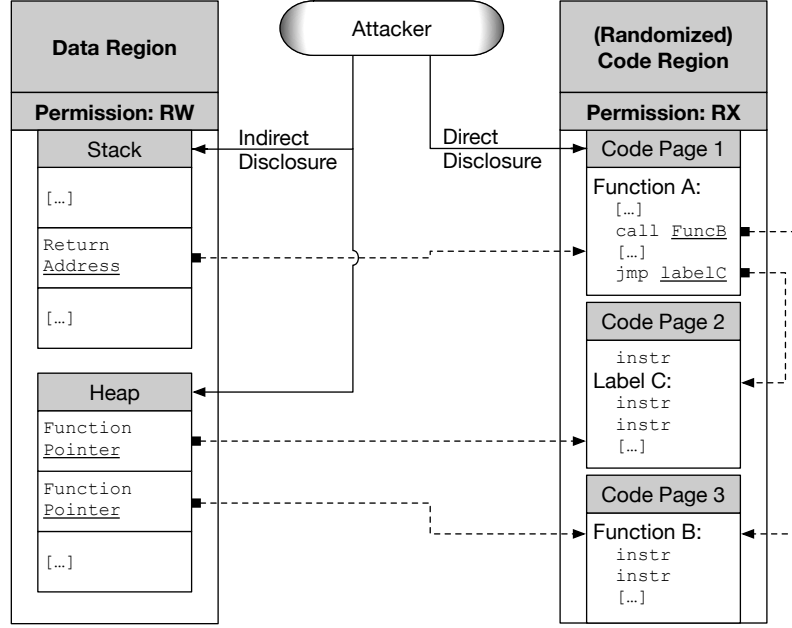


Figure 14: Direct and indirect memory disclosure.

3.1.1 Threat Model

For our attacks, we assume the following threat model which is in-line with previous offensive work [192].

Defense Capabilities

Writable \oplus Executable Memory. We assume that throughout the execution $W\oplus X$ memory is enforced by the operating system. Hence, the attacker can neither inject new code nor modify existing code.

(Fine-grained) Code Randomization. We assume that the targeted application is protected with (fine-grained) code randomization. We do not make any assumptions about the granularity of the code-randomization scheme. In fact, the granularity might range from base address randomization [97, 150, 218], to function permutation [18, 46, 113], to basic block [227], to instruction randomization [162].

Just-in-Time (JIT)-Protection. We assume that the JIT code is hardened against JIT-spraying attacks [24]. Modern JIT-code compilers implement techniques like constant blinding and NOP [94] insertion to fulfill this requirement [13].

Adversary Capabilities

Memory-corruption Vulnerability. We assume the presence of a memory-corruption vulnerability, which enables the attacker to read and write arbitrary memory. Further, the vulnerability can be exercised multiple times consecutively without crashing the application.

Initial Information Disclosure. We assume the attacker obtained a *single* valid code pointer during run time.

Computation Engine. We assume that the attacker can perform arbitrary, but sandboxed, computations during run time. This assumption is satisfied by script engines that are embedded into many modern applications such as browsers, or document viewers.

3.1.2 Just-in-Time Return-oriented Programming: Direct-disclosure Attacks

Just-in-Time Return-oriented Programming (JIT-ROP) is a generic direct-memory disclosure attack framework that can bypass *any* code-randomization scheme. The JIT-ROP framework builds upon the assumption that the attacker can exercise an information-disclosure vulnerability multiple times and can access a computation engine during the attack. This assumption is easily fulfilled by targets upon which the attacker can utilize an embedded scripting engine, as it is the case for browsers or document viewers. JIT-ROP attacks can also be launched against server applications, e.g., web servers, as the attacker can do the necessary computation locally.

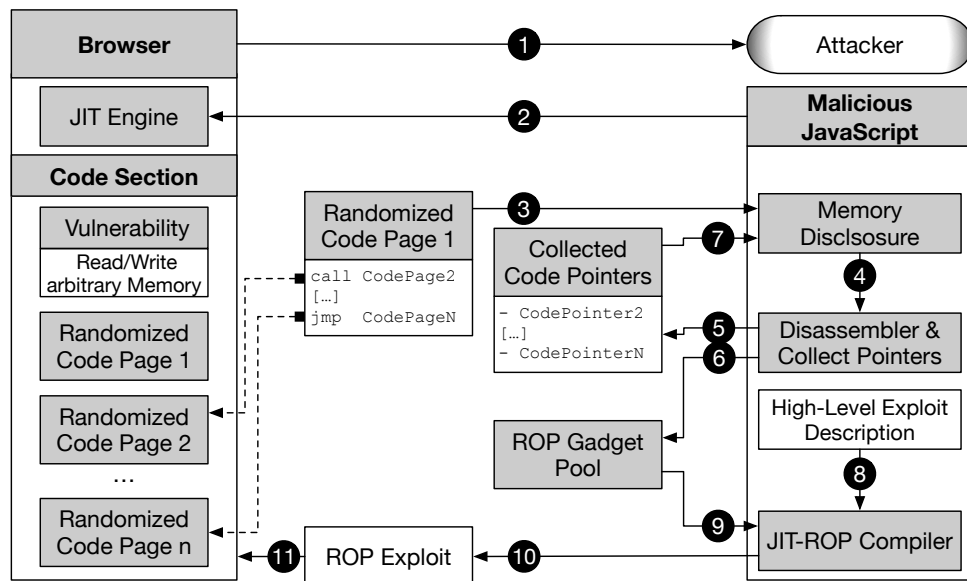


Figure 15: Detailed workflow of a JIT-ROP attack.

3.1.2.1 Attack Description

The main idea of JIT-ROP is to perform an analysis during run time of the randomized code, and to create an attack payload that is customized to the deployed code-randomization scheme. The JIT-ROP attack framework takes three arguments as input: (1) a read-write primitive which is achieved by exploiting a memory-corruption vulnerability. (2) a pointer to a valid instruction within the code region; (3) an attack

payload which is written in a high-level language. Unlike regular return-oriented programming (ROP) attacks JIT-ROP does not require any knowledge about the kind of code pointer nor to where it points to. As such JIT-ROP can take any code pointer, e.g., a return address or a function pointer. Based on this input JIT-ROP dynamically discloses the content of multiple code pages, searches for gadgets, and compiles the high-level attack description into a concrete ROP attack based on the found gadgets. In the forthcoming pages we describe in detail the workflow of a JIT-ROP attack with the help of Figure 15. In our example, the attack is executed against a browser. However, JIT-ROP is not limited to browsers but can be launched against every client-side application that features a scripting engine, like document viewers, flash or word processors, or server-side applications like web servers.

In the attack scenario, depicted in Figure 15, the attacker first lures the victim to visit an attacker controlled website (Step ❶). Through the website the attacker serves a malicious JavaScript program which exploits a memory-corruption vulnerability of the browser (Step ❷) to corrupt its internal data structures. This corruption enables the attacker to read and write arbitrary memory. Next, the attacker utilizes the read access to disclose the content of the memory page to which the initially disclosed code pointer points to (Step ❸). This is possible because memory pages are 4KB aligned. The content of the disclosed page is then analyzed with the help of disassembler of the attack framework (Step ❹). The disassembler has two tasks: first, it disassembles the code of the current code page to extract new code pointers which are encoded into direct branch instructions, e.g., call or jump instructions (Step ❺). Some of the extracted code pointers will reference other code pages. By recursively disassembling the newly discovered code pages JIT-ROP is able to gradually disclose all code pages of the targeted application (Step ❽). Second, the disassembler searches each disclosed memory page for useful ROP gadgets (Step ❻). Once JIT-ROP found all ROP gadgets, which are required to assemble an attack payload, it stops disclosing code page. In the final step, JIT-ROP takes the high-level attack description, and the disclosed ROP gadgets as an input for the attack compiler to generate a ROP payload, which is customized to the deployed code-randomization scheme (Steps ❸–❿). Finally, the attacker uses the write access to overwrite a code pointer to hijack the control flow and to execute the customized ROP payload.

3.1.2.2 *Lessons learned*

For our evaluation, we implement JIT-ROP by exploiting a heap-based buffer overflow (CVE-2012-1876) in Internet Explorer 8 on Windows 7 [224]. We found that JIT-ROP discloses the content of 301 memory pages before it gathers the required gadgets to compile and execute our payload which first resolves Windows Application Programming Interface (API) functions to then start the Windows calculator.

JIT-ROP dynamically adapts attack payloads to the memory layout of the targeted application, and hence, bypasses all code-randomization techniques that aim at mitigating code-reuse attacks. Thus, to defeat JIT-ROP attacks, code randomization must be protected against information-disclosure attacks. Backes and Nürnberger [16] proposed an initial approach which obfuscates the code pointers of direct

branch instructions. However, as we will show next (Section 3.1.3), mitigating information-disclosure attacks is far more complex than initially anticipated.

3.1.3 *Isomeron: Indirect-disclosure Attacks*

JIT-ROP makes no assumption about the deployed code-randomization scheme but relies on a scripting engine to perform the analysis of the protected binary during run time. However, in practice it is safe to assume that the attacker has exact knowledge of the deployed mitigations.

Contrary to direct disclosure attacks, like JIT-ROP, *indirect* disclosure attacks do not need to read the code section of the protected binary but solely rely on gathering code pointers from data memory. By combining the disclosed code pointers with an offline analysis of the targeted application, and knowledge of the deployed code-randomization technique, the attacker can infer a Turing-complete gadget set. For example, let us assume that a fine-grained randomization scheme, which permutes the order of the functions in memory is chosen to mitigate code-reuse attacks. Further, direct read access to the code sections is prevented (we elaborate on this topic in Chapter 4). Through an offline analysis the attacker first extracts a mapping between all available gadgets within the target application and the function that contains the gadget. Next, the attacker compiles an attack payload based on the available gadgets, and consults the extracted mapping to discover which functions include the used gadgets. During run time, the attacker then learns the addresses of these functions by means of indirect information disclosure. Since the applied code randomization only changes the order of the functions, but not their content, the gadgets within the functions stay intact. Hence, the attacker only needs to disclose the address of the function, which contains the required gadgets, after the code was randomized to conduct a code-reuse attack.

In practice, indirect information-disclosure attacks are powerful enough to bypass code-randomization schemes because the data memory contains a large number of code pointers. The number of code pointers, which the attacker needs to disclose for a successful attack, greatly depends on the granularity of the deployed code-randomization scheme, i.e., the more fine granular the randomization the more code pointers are required. However, the attacker can increase the number of available code pointers in the data memory by carefully choosing the input to the targeted application in order to trigger the execution of code paths which write the required code pointers to memory.

In the following section we provide a detailed description of how indirect disclosure improves JIT-ROP to bypass Oxymoron [16], which was the first attempt to mitigate JIT-ROP attacks.

3.1.3.1 *Beyond Fine-grained ASLR: Bypassing Oxymoron*

Recently, several [16, 17] code randomization schemes have been proposed that aim at tackling JIT-ROP. However, Oxymoron [16] was the first published approach that claims

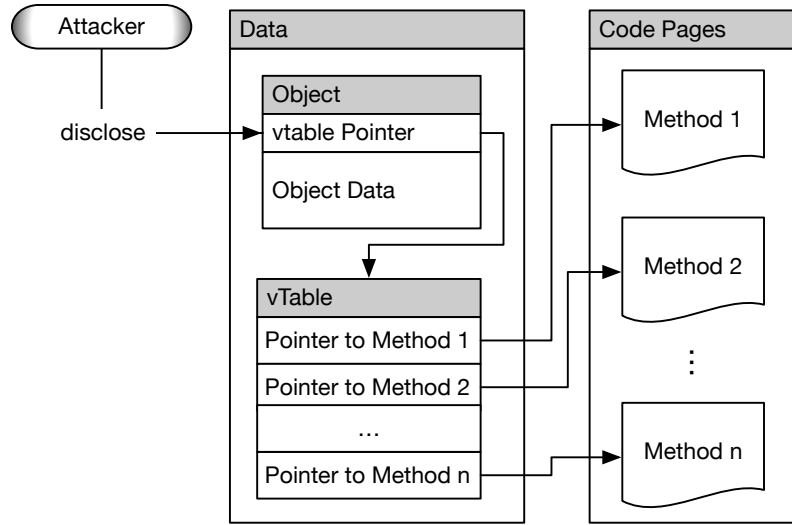


Figure 16: Example of how disclosing a vtable pointer allows the attacker to identify valid mapped code pages.

to resist JIT-ROP. Hence, we conducted a security analysis of Oxymoron, and extended JIT-ROP to successfully bypass it.

The main goal of Oxymoron is to (i) enable code sharing for randomized code, and (ii) hide code references encoded in direct branch instructions. The latter effectively prevents the attacker from discovering and disassembling new code pages (Step 5 in Figure 14), since the attacker can no longer follow a direct branch target to identify a new mapped page. Internally, Oxymoron uses a combination of page-based randomization and x86 segmentation to reach its goals. For this, Oxymoron transforms direct inter-page branches into indirect branches. The original destination addresses of all transformed branches are maintained in a special and hidden table. Specifically, the table is allocated at a random location in memory and Oxymoron assumes that the attacker cannot disclose the location and content of this table. In particular, Oxymoron forces the transformed branch instructions to address the table through a segment register which holds an index to the table. The use of a segment register creates an indirection layer that cannot be resolved by attackers in user-mode, because the information necessary for resolving the indirection are maintained in kernel space. While Oxymoron indeed hinders JIT-ROP from discovering new code pages, we show in the following that the Steps 4 - 7 in Figure 14 can be easily modified and bypass Oxymoron's protection. To demonstrate the effectiveness of our new technique, we developed an exploit targeting Internet Explorer 8 which bypasses Oxymoron.

3.1.3.2 High-level Attack Description

The main weakness of Oxymoron concerns the fact that it focuses only on hiding code pointers encoded into direct branches. However, disassembling code pages and following direct branches to new pages, is only one way of discovering addresses of new code pages. Using *indirect* memory disclosure, the attacker can leverage code pointers

stored on the stack and heap to efficiently disclose a large number of code pages and ultimately launch a JIT-ROP attack.

Code pointers of interest are return addresses, function pointers, as well as pointers to virtual functions which are all frequently allocated on data memory. In case of programs developed in object-oriented programming languages like C++, one obvious source of information are objects which contain virtual functions. In order to invoke these virtual functions, a vtable is used. This table is allocated as an array containing the target addresses of all virtual functions. Since vtables are frequently used in modern applications, and since their location can be reliably determined during run time, we exploit them in our improved JIT-ROP attack. Nevertheless, code pointers on the stack, such as return addresses, can be also leveraged in the same manner for the case the target application does not populate any vtables.

As shown in Figure 16, the first step of the attack is to disclose the address of the so-called vtable pointer which subsequently allows the attacker to disclose the location of the vtable. Once the virtual function pointers inside the vtable are disclosed, the attacker can determine the start and end address of those pages where virtual functions reside. For a target application such as a web browser or a document viewer, it is very likely to find complex objects with numerous function pointers. A large number of function pointers increase the number of valid code pages whose page start and end the attacker can reliably infer. Given these code pages, the attacker can then perform Step ⑥ to ⑪ as in the original JIT-ROP attack.

In the following, we apply our ideas to a known vulnerability in Internet Explorer 8, where we assume Oxymoron's protection mechanisms to be in-place.¹ Specifically, we take an existing heap-based buffer overflow vulnerability (CVE-2012-1876 in Internet Explorer 8 on Windows 7, which is well-documented [224]). We exploit this vulnerability to validate how many code pages attacker may identify using our above introduced techniques, and whether this code base is sufficiently large to launch a reasonable code reuse attack.

3.1.3.3 Exploit Implementation

As in any other code reuse attack, we require the target application to suffer from (i) a memory error (buffer overflow), and (ii) a memory disclosure vulnerability. The former is necessary to hijack the control-flow of the application, and the latter to disclose the vtable pointer which is the starting pointer to launch our attack (see Figure 16).

An additional requirement for our attack is the identification of C++ objects in Internet Explorer that populate virtual tables, i.e., contain many virtual functions. For this, we reverse-engineered C++ objects in Internet Explorer and identified several complex objects containing a large number of virtual functions (see Table 2). Once we are aware of the main target C++ objects, we can pick one (or more), and write a small JavaScript program that allocates our target object on the heap.

The next step is to dynamically read the vtable pointer of the target C++ object at run time. However, this raises a challenge as ASLR randomizes code and data segments.

¹ Note that Oxymoron's source code is not public. Hence, we simply assume its protection is active.

C++ Object	Virtual Functions
CObjectElement	150
CPluginSite	150
CAnchorElement	146
CAreaElement	146
CHyperlink	146
CRichtext	144
CButtonLayout	144
...	...

Table 2: Excerpt of C++ objects in Internet Explorer containing a large number of virtual functions

The run time location of the vtable pointer is not per-se predictable. However, due to the low randomization entropy of ASLR for data segments, the relative address (offset) to another memory object is in most cases predictable.

Hence, in order to exploit this circumstance, the attacker needs to allocate the target C++ object close to an *information-leak object* such as a JavaScript string. Carefully arranging objects close to each other to perform memory disclosure is commonly known as *heap feng shui* [201]. In fact, we re-use this attack technique and arrange objects using JavaScript as shown in Figure 17.

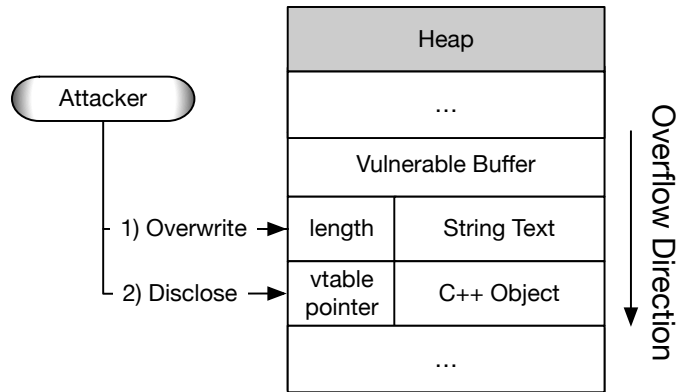


Figure 17: Heap-Layout of our Exploit.

Specifically, we allocate via JavaScript a buffer, a string, and our target C++ object which contains many virtual functions. The string object consists of two fields, namely, the string length field holding the size of the string, and the string text itself. The memory error in Internet Explorer allows us to overflow the vulnerable buffer. As the string object is just allocated next to the vulnerable buffer, our overflow overwrites the string length field with a value of the attacker's choice. As we set the value to its maximum size (i.e., larger than the actual string length), we are able to read beyond the string boundaries. Since our C++ object (in our exploit the CButtonLayout object) is just allocated next to the string, we can easily disclose its vtable pointer. Afterwards, we follow the vtable pointer to disclose all functions pointers of our C++ object.

Note that Figure 17 actually contains a simplified view of our target C++ object `CButtonLayout`. By disassembling (see Listing 2) the function which creates the `CButtonLayout` object, we recognized that this C++ object contains two vtable pointers. Altogether with these two vttables we could extract 144 function pointers, and hence 74 unique code pages. In our particular exploit, the number of code pointers resp. unique pages could be increased to 322 resp. 87 pages due to the fact that the page where the two vttables of the `CButtonLayout` object reside, contains two additional vttables of other C++ objects. The attacker can always increase the number of leaked vttables by allocating more complex objects (as given in Table 2) on the heap.

```

1  push    0xFCh          ; dwBytes
2  push    8              ; dwFlags
3  push    _g_hProcessHeap ; hHeap
4  call    ds:HeapAlloc(x,x,x)
5  mov     esi, eax
6  [...]
7  mov     dword ptr [esi], offset const CButtonLayout::`vftable' {for `CLayoutInfo'}
8  mov     dword ptr [esi+0xC], offset const CButtonLayout::`vftable' {for `CDispClient'}

```

Listing 2: Disassembled code that creates the `CButtonLayout` object

The 87 leaked code pages give us access to a large code base (348 KB) for a code reuse attack. Hence, the next attack step involves gadget search on the 87 leaked code pages. For our proof-of-concept attack, we identified all gadget types (load, store, add) necessary to launch a practical return-oriented programming attack; including a stack pivot gadget [241]. One important gadget is a system call gadget to allow interaction with the underlying operating system. The original JIT-ROP attack leverages for the dynamic loader functions `LoadLibrary()` and `GetProcAddress()` allowing the attacker to invoke any system function of his choice. However, when the addresses of these two critical functions are not leaked (as it is the case in our exploit), we need to search for an alternative way. We tackle this problem by invoking system calls directly. On Windows 32-bit, this can be done by loading (i) the system call number into the `eax` register, (ii) a pointer to the function arguments into `edx`, and (iii) invoking a `syscall` instruction on our leaked pages. At this point, we are able to compile any return-oriented programming payload as our leaked code pages contain all the basic gadget types.

Specifically, we constructed an exploit that invokes the `NtProtectVirtualMemory` system call to mark a memory page where we allocated our shellcode as executable. We use a simple shellcode, generated by Metasploit [135] that executes the `WinExec()` system function to start the Windows calculator to prove arbitrary code execution.

The last step of our attack is to hijack the execution-flow of Internet Explorer to invoke our gadget chain. We can do that simply by exploiting the buffer overflow error once again. In contrast to the first overflow, where we only overwrote the string length field (see Figure 17), we overwrite this time the vtable pointer of our target C++ object, and inject a fake vtable that contains a pointer to our first gadget. Afterwards, we call a virtual function of the target C++ object which redirects the control-flow to our gadget chain (as we manipulated the vtable pointer).

3.1.3.4 *Lessons learned*

In summary, our attack bypasses Oxymoron as it discovers valid mapped code pages based on code pointers allocated in data structures (specifically, virtual function pointers). As Oxymoron only protects code pointers encoded in branch instruction on code segments, it cannot protect against our improved JIT-ROP attack. In order to defend against this attack, one also needs to protect code pointers allocated in data structures. Note that our attack is general enough to be applied to any other memory-related vulnerability in Internet Explorer, simply due to the fact that Internet Explorer contains many complex C++ objects with many virtual functions (see Table 2).

3.1.4 *Conclusion*

Fine-grained code-randomization defenses shuffle the memory layout of applications during run time. As a consequence, the attacker cannot rely on pre-computed addresses and offsets of code snippets, which are necessary for code-reuse attacks.

In this section, we introduced two advanced memory-disclosure attacks that highlight the conceptual weakness of randomization-based mitigations. Specifically, we show that the attacker can repeatedly exploit memory-disclosure vulnerabilities to either directly or indirectly disclose information about the randomized code. We adapt real-world exploits to leverage our attack techniques to bypass any code-randomization scheme.

Our attack techniques highlight the need of leakage resilience for code-randomization in order to provide an effective protection from sophisticated code-reuse attacks. In fact, we utilize the insights we gain in this section to design a leakage resilient code-randomization technique, which is based on execute-only memory, and which we present in Chapter 4.

3.2 LOSING CONTROL: ATTACKS ON FINE-GRAINED CFI

For the sake of efficiency and flexibility the C languages family eschew security features such as automatic memory management, strong typing, and overflow detection. As a result, programming errors can lead to memory corruption that causes unexpected program behavior routinely exploited by attackers—often with severe consequences.

Defending against exploits is extremely challenging. Any technique hoping for deployment in practice needs to minimize the performance impact, remain fully compatible with vast amounts of existing code, and require no manual effort to apply. Among the few defenses that meet this high bar are stack canaries, data execution prevention (DEP), and address space layout randomization (ASLR). On the other hand, the combination of these defenses has still not prevented sophisticated real-world attacks. For this reason, the security research community is exploring a number of potential successors to today’s standard mitigations.

The goal of this section is point out unforeseen weaknesses in recent defenses that are thought to be substantial advances over current mitigations. Control-flow integrity provides substantially better protection against control-flow hijacking than any mitigation in use today. Our analysis focuses on the so called fine-grained forward edge variant (FE-CFI) [215] because 1) a production-grade implementation is available and 2) recent research has demonstrated serious weaknesses in coarse-grained variants of CFI [31, 60, 83, 84]. As the name implies, FE-CFI does not protect backward control-flow edges from callee functions back to their callers. For this reason, FE-CFI must be paired with some return address protection mechanism. We examine two such mechanisms: StackGuard [48], which is a standard mitigation, and StackAmor [41] which represents the state-of-the-art in the area of stack protection mechanisms. Finally, we analyze XnR, an improvement over Data Execution Prevention (DEP). DEP implements the principle of least privilege for virtual memory pages. Under DEP, execution permissions, imply read permissions; XnR emulates execute-no-read permissions with the aim to prevent JIT-ROP attacks.

Based on our analysis of the above defenses, we introduce a new class of memory disclosure attacks that read and manipulate the stack to gain control of a vulnerable application. To demonstrate the threat of such attacks, we constructed a non-trivial, real-world exploit—which we named StackDefiler—that is capable of bypassing all of the defenses we examined.

Summing up, our contributions are:

- **Security analysis of state-of-the-art defenses.** Defenses based on CFI are in principle immune to memory disclosure because they do not rely on information hiding. The two other defenses explicitly advertise resistance to memory disclosure. In our analysis of these defenses, we found they all have weaknesses that are exploitable through memory disclosure and corruption of the stack.
- **Bypassing FE-CFI implementation.** To confirm that the weaknesses we identified are exploitable in practice, we use it to bypass Google’s Fine-Grained Forward CFI implementation. Specifically, we found that a critical CFI pointer is spilled to the

stack and can be corrupted by attackers to hijack the control flow. We removed this weakness and evaluate the performance impact of our fix.

- **Bypassing StackArmor.** StackArmor claims to provide temporal safety for stack contents. We demonstrate that by using multiple, malicious web workers (an HTML5 feature to allow concurrency in JavaScript), the attacker can discover the stack layout and overwrite control flow meta-data such as return addresses.
- **Discovering the Code Layout with XnR enabled.** XnR is intended to be used in conjunction with fine-grained ASLR such as ASLP. We show that applications protected by XnR and ASLP, we can control which return addresses are placed on the stack and harvest return addresses to indirectly disclose the code layout without directly reading the code.

3.2.1 Threat Model

Our threat model captures the capabilities of real-world attacks, and is in line with the common threat model of CFI [6], as well as with the prior offensive work [68, 186, 188, 196].

Defense Capabilities

Non-Executable Memory The target system enforces data execution prevention (DEP) [136]. Otherwise the attacker could directly manipulate code (e.g., overwriting CFI checks), or inject new malicious code into the data section of a program. The attacker is therefore limited to code-reuse attacks.

Randomization. The target system applies address space layout randomization (ASLR).

Shadow Stack. We do not have access to the implementation of shadow stacks [3, 56]. Therefore, we assume the presence of an adequate shadow stack implementation.

Adversary Capabilities

Memory read-write. The target program contains a memory-corruption vulnerability that allows the attacker to launch a run-time exploit. In particular, we focus on vulnerabilities that allow the attacker to read (information disclosure) and write arbitrary memory. Such vulnerabilities are highly likely as new vulnerabilities are being constantly reported. Common examples are use-after-free errors [214].

Adversarial Computation. The attacker can perform computations at run time. Many modern targets such as browsers, Flash, Silverlight, and document viewers, as well as server-side applications and kernels allow the attacker to perform run-time computations. Real-world attacks on client-side applications typically utilize a scripting environment to instantiate and perform a run-time exploit. Additionally, the attacker can use the scripting engine to generate multiple execution threads.

3.2.2 *StackDefiler*

Our attacks are based on modifying data on the stack. Hence, as a first step, in the presence of ASLR, we must disclose the address of the stack. We stress that we do not rely on *stack-based* vulnerabilities to attack the stack. Instead, we used *heap-based* vulnerabilities in our exploits. We observe that attackers with the ability to disclose arbitrary memory can get a stack address by recursively disclosing data pointers (see Section 3.2.3.2). Attacking values on the stack is challenging, because (i) only certain functions will write critical data to the stack, and (ii) the lifetime of values on the stack is comparatively short, i.e., generally during the execution of a function. Nevertheless, we are able to manipulate targeted values on the stack.

In the following we give a high-level description of our attacks. For this we discuss three different stack-corruption techniques that allow us to bypass the CFI implementations we examined.

3.2.2.1 *Corrupting Callee-Saved Registers*

To maximize the efficiency of a program, the compiler tries to maximize the use of CPU registers, instead of using the (slower) main memory. The compiler performs register allocation to keep track which registers are currently in use and to which it can assign new values. If all registers are in use, but a register is required to perform a computation, the compiler temporarily saves the content of the register to the stack. When a function (the caller) calls another function (the callee), the callee cannot determine which of the caller's registers are used at the moment of the call. Therefore, the callee saves all registers it needs to use during its execution temporarily on the stack. These saved registers are called *callee-saved registers*. Before the callee returns to the caller it restores all callee-saved registers. While the registers are saved, the attacker can change the values on the stack and therefore corrupt the callee-saved registers. This becomes a severe problem if the caller uses the restored (and potentially corrupted) registers for CFI checks and can affect all architectures where the application binary interface (ABI) specifies the concept of callee-saved registers.

We found that two CFI implementations, IFCC and VTV [215], are vulnerable to this kind of attack. As we will argue in the following, this threat becomes even more crucial for applications that are compiled with position-independent code (PIC) for architectures that do not support program-counter relative (PC-relative) addressing, such as x86 32-bit.

On systems like Mac OS and Linux, ASLR compatible binaries contain position-independent code (PIC). Position independence means that all code references are relative to the program counter (PC). This allows the dynamic loader to load the binary at an arbitrary base address without relocating it.

However, Intel x86 processors running 32-bit code do not directly support PC-relative addressing. As a workaround, PIC on x86 requires the program to obtain the current value (i.e., the absolute address) of the program counter dynamically at run time. Once this address is known, the program can perform PC-relative references. At assembly level this is implemented by executing a call to the subsequent instruction. The call

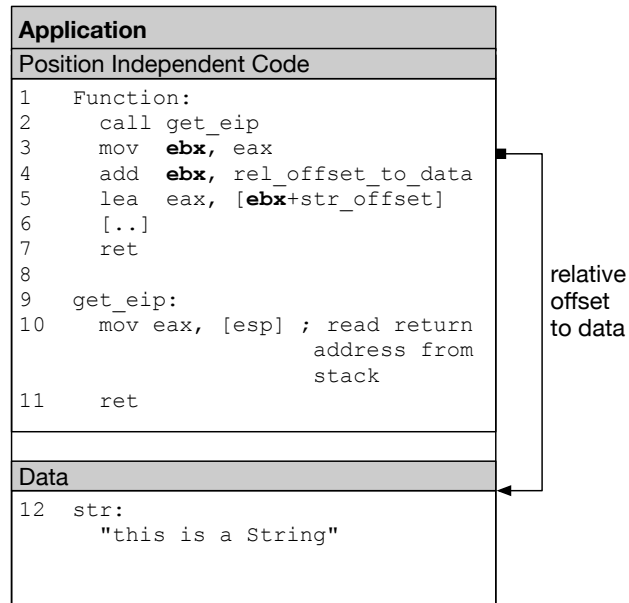


Figure 18: Application compiled with position-independent code. To get the absolute address of `str` the compiler emits instructions that first receive the absolute address of Function at run time. The absolute address of `str` is then calculated by adding the relative offset between Function and `str`, calculated by the compiler, to the absolute address of Function.

automatically loads the return address onto the stack, where the return address is simply the absolute address of the subsequent instruction. Hence, the program can obtain its current program counter by simply popping the return address off the stack in the subsequent instruction. Once the program counter is loaded into a register, an offset is added to form the position-independent reference.

Figure 18 illustrates how position-independent code references the global string variable `str` in the data section (line 12). At function entry, the function calls `get_eip()` (line 2). This function (line 9) only reads the return address from the stack (line 10), which is the address of the instruction following the call of `get_eip()` (line 3). Next, the result is moved into the `ebx` register (line 3). We noticed that both LLVM and GCC primarily use the `ebx` register to compute position-independent references (line 5).

Subsequently, the program can perform PC-relative addressing to access the global string variable: the `add` instruction adds the relative offset between the data section and the current function to `ebx` which now holds a pointer to the data section (line 4). Finally, the offset of the string within the data section is added to `ebx` and the result (address of the string variable) is saved in the `eax` register (line 5).

On x86 32-bit platforms PIC becomes a vulnerability for CFI, because the global CFI policies are addressed through the `ebx` register. Since `ebx` is a callee-saved register it is spilled on the stack by all functions that perform CFI checks.

3.2.2.2 Corrupting System Call Return Address

Fine-grained CFI as proposed by Abadi et al. [3] validates the target address of every indirect branch. Valid forward edges of the CFG are determined using static analysis and are enforced through label checking. A shadow stack is used to verify the backward edges of the CFG. We noticed that user-mode CFI only instruments user-mode applications and not the kernel. In general, this makes sense because the kernel isolates itself from user-mode applications, and hence, is considered trusted. However, we discovered a way to undermine this trust to bypass CFI without compromising the kernel. In particular, we exploit the fact that the kernel reads the return address used to return from a system call to the user mode from the user-mode stack.

On x86 32-bit a special instruction—`sysenter`—was introduced to speed up the transition between user and kernel mode [105]. The `sysenter` instruction does not save any state information. Therefore, Windows saves the return address to the user-mode stack before executing `sysenter`. After executing the system call, the kernel uses the saved return address to switch back to user mode. This opens a small window of time between the return address being pushed on the stack and the kernel reading it to switch back to user mode. We use a second, concurrent thread that exploits this window to overwrite the saved return address. Hence, when returning from a system call the kernel uses the overwritten address. This allows the attacker to set the instruction pointer to an arbitrary address and bypass CFI policy checks.

Note that this attack works within the threat model of CFI because we never modify existing code, nor corrupt the kernel, or tamper with the shadow stack, but we exploit a missing check of a code pointer that can be controlled by the attacker.

The 64-bit x86 architecture uses a different instruction, called `syscall`, to switch from user to kernel mode. This instruction saves the user-mode return address into a register, thus preventing the attacker from changing it. However, even 64-bit operating systems provide an interface for `sysenter` to be compatible with 32-bit applications. Hence, 32-bit applications that are executed in 64-bit operating systems remain vulnerable. Another pitfall of 64-bit x86 is that it partially deprecates memory segmentation, hence, the shadow stack can no longer be completely protected via hardware. As a consequence, the protection of the shadow stack relies on information hiding or less efficient software-fault isolation techniques.

3.2.2.3 Disclosing the Shadow Stack Address

Dang et al. [56] survey the different implementations of shadow stacks and their performance costs. One observation is that a *parallel shadow stack*, i.e., a shadow stack located at a constant offset to the normal stack, provides the best performance. However, as we demonstrate in Section 3.2.3.2 the attacker can leak the address of the normal stack and therefore compute the address of the shadow stack.

Another shadow stack technique utilizes the *thread-local storage* (TLS), a per-thread memory buffer usually used to store thread-specific variables. In the following we discuss potential implementation pitfalls of this approach. However, we have not implemented this attack due to the unavailability of implementations in public domain.

TLS is addressed through a segment register. Although segmentation is no longer available under x86 64-bit, segment registers are still present and can be used to address memory. In general, a TLS-based shadow stack implementation first loads the shadow-stack pointer into a general-purpose register. Next, this general-purpose register is used to save the return address on the shadow stack [3, 56]. However, we did not find any evidence that the general-purpose registers used during this operation are cleared afterwards. Hence, the address of the shadow stack may be leaked when a function pushes the used register on the stack. Further, an application might hold a reference to TLS in one of its memory objects that can be leaked to disclose the memory address of TLS and the shadow stack.

3.2.3 *Attack Implementations*

We now turn our attention to the practical implementation of the previously described attacks. To prove the effectiveness of these attacks we start from real-world vulnerabilities. For our proof-of-concept implementation of the attacks we chose the Chromium web browser because it is available for all common operating systems, and implements state-of-the-art heap and stack software defenses. We stress that our attacks also apply to other applications that provide the adversarial capabilities we outlined in Section 3.2.1. This includes document viewers, Flash, Silverlight, server-side applications and kernels. We re-introduced an older software vulnerability (CVE-2014-3176) in the most recent version of Chromium (v44.0.2396.0)—we did not make any further changes to the source code.

To prove that stack spilled registers pose a severe threat to modern, fine-grained forward-edge CFI implementation we compiled Chromium with IFCC for 32 and 64-bit on Ubuntu 14.04 LTS. We disassembled IFCC and VTV protected applications to verify that they are vulnerable to stack-spilling attacks on other operating systems (Unix and Mac OS X) as well. We implemented our attack against the initial proposed CFI [3] on a fully patched Windows 7 32-bit system. Since the implementation of the originally proposed CFI [3] is not available, we assume that fine-grained CFI with a secure shadow stack deployed and construct our attack under the constraints given by the chapter.

After giving a short introduction to browser exploitation, we give a detailed description of our proof-of-concept exploits that bypass existing CFI implementations.

3.2.3.1 *Attacking a Web Browser*

While attacker-controlled JavaScript in browsers is generally sandboxed by enforcing type and memory safety, the runtime used to interface the browser and web contents is not. Performance critical parts of the JavaScript runtime library are written in lower level, unsafe languages, e.g., C++. The usage of C++ opens the door for memory-related security vulnerabilities. Memory corruption is then used to manipulate the native representation of website objects, which cannot be done directly from JavaScript code. Next, we explain how this can be exploited to read arbitrary memory and hijack the program control flow.

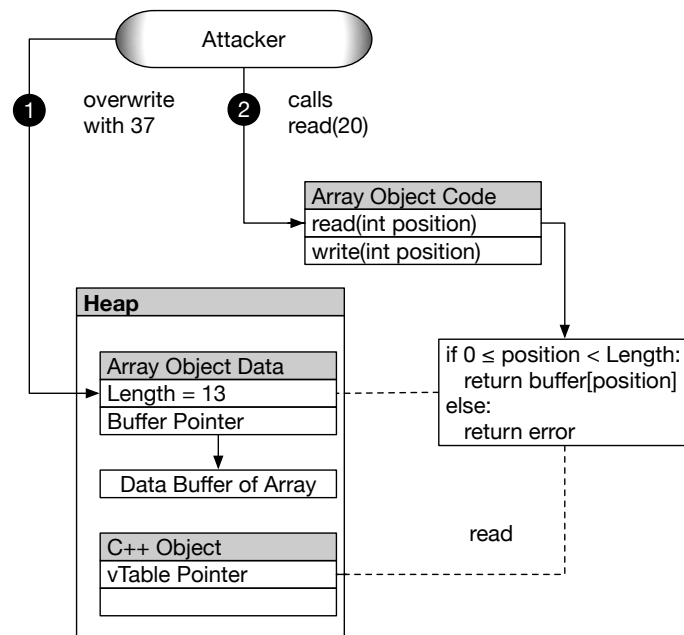


Figure 19: The attacker can overwrite the length field of an array object. He uses the native read function to disclose memory content beyond the array buffer, e.g., the vTable pointer of a consecutive object.

Information Disclosure

Websites create a variety of objects using the browser's scripting engine. These objects are stored consecutively in memory. For instance, the native representation of an array object is usually a C++ object with two fields: the length of the array followed by its starting address, as shown in Figure 19. A JavaScript program can read the contents of the array by using the runtime interface provided by the native C++ object. To ensure memory safety, the native read function uses the saved array length to ensure that the JavaScript program does not access memory outside the arrays bounds. By using a memory corruption vulnerability, the attacker can overwrite the array length in the native representation of the array object with a larger value, as shown in Step ①. This allows the attacker to read the memory beyond the original array boundaries using normal JavaScript code (Step ②) and disclose the contents of a subsequent C++ object.

vTable hijacking

To hijack the program's control flow, the attacker must overwrite a code pointer holding the destination of an indirect branch instruction. C++ virtual function tables (vTables) are commonly used for this purpose. The vTable is used to resolve virtual functions call targets at run time and contains an array of pointers to virtual functions, along with other data. The entries of a vTable cannot be overwritten because they reside in read-only memory. However, each C++ object that uses virtual functions maintains a pointer to its corresponding vTable. Since this pointer is a field of the object, it is stored in writable memory. The attacker can exploit a memory corruption vulnerability

to overwrite the vTable pointer of a C++ object with a pointer to a fake vTable which he created and injected beforehand. Instead of the original table of function pointers, all function pointers in the fake vTable will point to the code the attacker aims to leverage for a code-reuse attack. Lastly, after overwriting the vTable pointer of an object, the attacker uses JavaScript code interfaces to the native object to invoke a virtual function from the fake vTable.

3.2.3.2 Proof-of-Concept Exploit

Our exploit performs the following steps: (i) Gain arbitrary read and write capabilities, (ii) locate the stack and disclosing its contents, and (iii) bypass the CFI check and hijack the control flow.

The re-introduced vulnerability (CVE-2014-3176) allows us to manipulate the data fields of JavaScript objects on the heap, such as ❶ in Figure 19. Once an array-like object has been corrupted, we can access adjacent memory location without failing a bounds check (see ❷ in Figure 19). In our exploit, we use the corrupted object to manipulate the buffer pointer field of a JavaScript ArrayBuffer instance. By setting the buffer pointer to the address we want to access, we can then read and write arbitrary memory by accessing the first element of the ArrayBuffer via the JavaScript interface. There are many ways to corrupt array-like objects, hence, our exploit does not depend on a specific type of memory corruption vulnerability.

Disclosing Data Structures

Chromium places different memory objects in different heaps. For instance, the array instance in Figure 19 is stored in the *object heap* while the data buffer it contains is in the *buffer heap*. The use of separate heaps prevents exploit techniques such as *heap feng shui* [201] which the attacker has used to co-locate vulnerable buffers and C++ objects [224].

However, during the analysis of Chromium's heap allocator, we found a way to force the allocator to place the vulnerable buffer at a constant offset to metadata that is used by the allocator to manage the different heaps. Chromium's heap allocator, *PartitionAlloc*, pre-allocates memory for a range of different buffer sizes. However, when memory for a buffer is requested that was not pre-allocated, *PartitionAlloc* will request memory from the operating system. Since *PartitionAlloc* needs to manage the dynamically allocated memory buffers, it requests two additional, consecutive memory pages from the operating system. The newly requested memory is organized as follows:

- (i) Meta information of allocated memory. This includes a pointer to the main structures of *PartitionAlloc*, which contains all information to manage existing and future allocations.
- (ii) Guard page. This page is mapped as inaccessible, hence, continuous memory reads/writes will be prevented. However, it does not prevent non-continuous reads/writes.

- (iii) Memory to fulfill allocation request. This is the memory that is used by *PartitionAlloc* to allocate buffers.

By allocating a large buffer (e.g., 1MB) which is very unlikely to happen during normal execution, we ensure that *PartitionAlloc* will allocate a new structure as previously described. We further know that the requested buffer will be placed at the start of (3), because it is the first buffer of this size. Since the offset between (i) and (iii) is constant, we can disclose the pointer to the main meta-data structure of *PartitionAlloc*. This allows us to identify all memory addresses used by the heap allocator, as well as predict which memory addresses will be used for future allocations.

This is a very powerful technique as we can predict the memory address of every C++ object that is created. Further we can control which objects are created at run time via the JavaScript interface. Hence, it becomes very hard to hide information (e.g., a shadow stack address) because as long as any object contains a pointer to the hidden information, we can disclose the information by creating the object and disclosing its memory.

Finally, in our attack, we choose to allocate an object that contains a vTable pointer, i.e. the *XMLHttpRequest* object. By overwriting the vTable pointer of this object with a pointer to a fake vTable, we can hijack the control flow (see Section 3.2.3.1).

Disclosing the stack address

To disclose and corrupt values on the stack to bypass CFI checks, we must first locate the stack in memory. In contrast to the heap, objects on the stack are only live until the function that created them returns. Hence, it is challenging to find a pointer to a valid stack address within the heap area. However, we noticed that Chromium's JavaScript engine, V8, saves a stack pointer to its main structure when a JavaScript runtime library function is called. Since the *ArrayBuffer.read()* function, which we use for information disclosure, is part of the runtime library, we can reliably read a pointer that points to a predictable location on the stack. The remaining challenge is to find a reference to a V8 object, because V8 objects are placed on a different heap than Chromium's objects. Hence, we need to find a reference from an object whose address we already disclosed to the V8 object that stores the stack address. We chose *XMLHttpRequest*, because it contains a pointer to a chain of other objects which eventually contain a pointer to the V8 object. Once we disclose the address of this object, we can disclose the saved stack pointer.

At this point we have arbitrary read and write access to the memory and have disclosed all necessary addresses. Hence, we now focus on implementing the attacks described in Section 3.2.2.

Bypassing IFCC

IFCC implements fine-grained forward-edge CFI and is vulnerable to attacks that overwrite registers which are spilled on the stack. For brevity, we omit the bypass of VTV. However, from a conceptual point of view there is no difference between the IFCC bypass and the one for VTV. Tice et al. [215] assume that the stack is protected by

StackGuard [48] which implements a canary for the stack to prevent any stack attacks. In practice, this does not prevent the attacker from overwriting the return address. Since IFCC focuses on the protection of CFG forward edges, we assume an ideal shadow stack to be in place that cannot be bypassed, though this might be hard to implement in practice.

IFCC protects indirect function calls by creating, at compile time, a list of functions that can be reached through indirect calls. It then creates a trampoline, i.e., a simple jump instruction to the function, for every function in this list. The array of all trampolines is called *jump table*. Finally, every indirect call is instrumented so it can only target a valid entry in the jump table.

Listing 3 contains the disassembly of an instrumented call. In the line 8 and 9, the target address of the indirect call and the address of the jump table are loaded into registers. Subtracting the base address of the target pointer and then using a logical *and* is an efficient way of ensuring that an offset within the jump table is used. Finally, this offset is added again to the base address of the jump table. This ensures that every indirect call uses the jump table, unless the attacker can manipulate the `ebx` register. As we explained in Section 3.2.2.1 `ebx` is a callee-saved register and therefore spilled on the stack during function calls.

For our exploit we target a protected, virtual function call F_{target} that is invoked (line 16) after another function F_{spill} is called (line 6), see Listing 3. During the execution of F_{spill} the `ebx` register is spilled on the stack (line 19): we overwrite both the target address of F_{target} through `vTable` injection (see Section 3.2.3.1) and the saved `ebx` register. We overwrite the saved `ebx` register such that line 9 will load the address of our gadget. After F_{spill} finishes execution, the overwritten register is restored and used to verify the subsequent call in F_{target} . The check will pass and line 16 will call our first gadget. After the initial bypass of CFI, we use unintended instructions to avoid further CFI checks.

Although 64-bit x86 offers more general-purpose registers, our analysis of a 64-bit, IFCC-protected Chromium version exposed that around 120000 out of 460000 indirect calls CFI checks (around 26%) are vulnerable to our attacks. We did not manually verify if all of these CFI checks are vulnerable. However, for a successful attack it is sufficient that only one of these CFI checks is vulnerable to our attack. We exploited one vulnerable CFI check to implement a similar attack and bypass IFCC for the 64-bit version of Chromium.

Bypassing fine-grained CFI

It seems that overwriting a user-mode return address used by a system call is straightforward. However, we encountered some challenges during the implementation. The first challenge is being able to correctly time the system call and the overwrite of the return address. We found the most reliable way is to spawn two threads: one thread constantly makes the system call and the other constantly overwrites the return address. The attack succeeded in 100% of our tests without any noticeable time delay.

```

1 F0:
2     call F0_next
3 F0_next:
4     pop ebx                ; load abs. address of F1
5     ; [...]
6     call F_spill
7     ; [...]
8     mov edi, [eax+4]       ; load address F_target
9     mov eax, [ebx-149C8h]  ; load jump-table
10    mov ecx, edi
11    sub ecx, eax           ; get offset in jump table
12    and ecx, 1FFFF8h      ; enforce bounds
13    add ecx, eax           ; add base addr jump table
14    cmp ecx, edi           ; compare target address
15    jnz cfi_failure
16    call edi               ; execute indirect call
17
18 F_spill:
19    push ebx
20    ; [...]                ; overwrite of ebx happens here
21    pop ebx
22    ret

```

Listing 3: Disassembly of an indirect call that is instrumented by IFCC.

```

1 ntdll!ZwWaitForSingleObject:
2     mov eax,187h          ; System call number
3     mov edx,offset SystemCallStub
4     call [edx]             ; call KiFastSystemCall
5     ret 0Ch
6
7     ; [...]
8
9 ntdll!KiFastSystemCall:
10    mov edx,esp
11    sysenter

```

Listing 4: ZwWaitForSingleObject System Call on Windows 7 32-bit.

We can utilize the *Web Worker* HTML5 API [228] to create a dedicated victim thread. During our analysis to find a suitable function that eventually invokes a system call, we noticed that an idle thread is constantly calling the `ZwWaitForSingleObject` system call which is shown in Listing 4. Line 4 shows the call that pushes the return address on the stack that is later used by the kernel to return to user mode.

Another challenge is that the constant invocation of the system call might corrupt any ROP gadget chain we write on the stack. Hence, we overwrite the user-mode return address with the address of a gadget which sets the stack pointer to a stack address that is not constantly overwritten. From there on we use gadgets that are composed of unintended instructions [192] to bypass the instrumented calls and returns.

This exploitation technique can bypass any fine-grained CFI solution that aims to protect 32-bit applications on Windows. This includes the initial CFI approach by Abadi et al. [3].

3.2.4 Mitigations

We consider possible mitigation techniques against our attacks. First, we describe our compiler patch for the IFCC/VTV implementation vulnerability and measure its performance impact on the SPEC CPU 2006 benchmarks. Subsequently, we discuss the broader problem of protecting the stack against memory disclosure and corruption attacks.

3.2.4.1 Patching IFCC

Recall that IFCC uses the base register containing the address of the GOT to reference the jump table validating the target of an indirect call (see Section 3.2.2.1). To prevent our attack presented in Section 3.2.3.2, we developed a compiler patch that safely reloads the GOT register before loading the CFI jump table. Our patch adds new instrumentation before the CFI check so this register is always re-calculated instead of being restored from the stack. With our proposed fix, IFCC uses three more instructions to validate each target which brings the total number of added instructions up to 15 per indirect call. Listing 5 shows an example of the IFCC instrumentation without our patch, and Listing 6 shows the reload we add on lines 12-17.

We measured the performance impact of this change using the SPEC CPU 2006 benchmark suite on a dual channel Intel Xeon E5-2660 server running Ubuntu 14.04 with Linux kernel 3.13.0. We selected only the benchmarks that have indirect calls since IFCC will not affect code that only uses direct calls. The benchmark results we report are medians over three runs using the reference inputs.

We report overheads relative to a baseline without IFCC enabled. Since IFCC uses link-time optimization, we also compile the baseline with link-time optimization turned on. Figure 20 shows that our patched version of IFCC performs between 0.12% and 1.19% slower (0.46% on average) than unpatched IFCC. Tice et al. [215] also found cases where IFCC outperforms the baseline, and we did not analyze these cases further. The patch for the 64-bit version is similar and was omitted for brevity.

We reported the weaknesses in IFCC and VTV and our mitigation for IFCC to the original developers of these mitigations.

3.2.4.2 Securing Stack

The machine stack is difficult to simultaneously secure against all types of attacks, since it must be readable and writable by the program. Similar to other exploit mitigation schemes, stack protection schemes can be categorized into schemes that rely on applying randomization to the stack or ensuring the integrity of the stack through isolation. We found that current stack randomization schemes introduce a lower performance overhead but remain vulnerable to our attack as the randomization secret can be disclosed, as we will discuss in the next section. On the other hand, isolating the stack

```

1  ; store current eip in ebx
2  call .next
3  .next:
4  pop ebx
5  add ebx, GLOBAL_OFFSET_TABLE
6  ...
7  ; call function which stores ebx to the stack
8  ...
9  ; Load destination function address
10 lea ecx, vtable+index
11 ; Load jump table entry relative to ebx
12 mov eax, [ebx + _jump_table.@GOT]
13 <perform CFI-check>
14 call ecx

```

Listing 5: Example IFCC assembly before fix

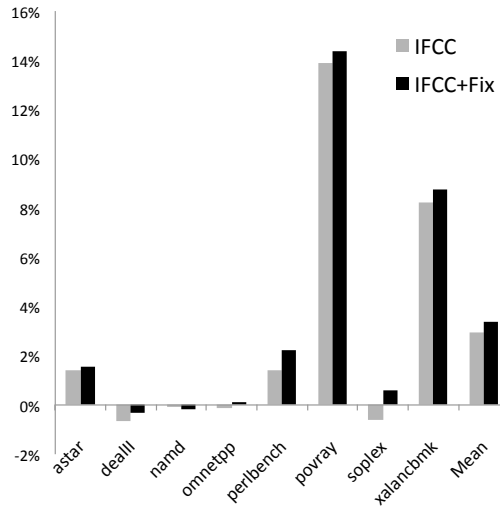


Figure 20: SPEC CPU2006 performance of IFCC-protected programs before and after we applied our fix relative to an unprotected baseline.

can potentially mitigate our attacks. However, current stack mitigation techniques are either not effective or suffer from non-negligible performance overheads.

Next, we shortly discuss the effectiveness of these mitigation schemes under our threat model.

Randomization-based Defenses

StackGuard [48] attempts to prevent stack-based buffer overflows by inserting a random stack cookie between potentially vulnerable buffers and the return address. However, this defense is insufficient against current attackers. An attacker with the capability to read the stack, as we have demonstrated with our attacks, can read and forge this cookie, even without an arbitrary memory write vulnerability.

The recently proposed StackArmor [41] further protects the stack not only from buffer overflows, but also from, but also stack buffer over/under reads and stack-based temporal vulnerabilities. However, StackArmor’s protections are confined to the stack

```

1      ; store current eip in ebx
2      call .next
3  .next:
4      pop ebx
5      add ebx, GLOBAL_OFFSET_TABLE
6      ...
7      ; call function which stores ebx to the stack
8      ...
9      ; Load destination function address
10     lea ecx, [vtable+index]
11
12     ; PATCH: Reload ebx with current eip, instead of
13     ;   untrusted, corruptible value
14     call .next2
15  .next2:
16     pop ebx
17     add ebx, GLOBAL_OFFSET_TABLE
18
19     ; Load jump table entry relative to ebx
20     mov eax, [ebx + _jump_table.@GOT]
21     ;
22     ; perform CFI check
23     ;
24     call ecx

```

Listing 6: Example IFCC assembly after fix

itself. Without any heap protection, an attacker can use heap-memory corruption to read and write arbitrary memory locations and can disclose metadata used by the StackArmor allocator to find and modify the stack.

Isolation-based Defenses

One possible mitigation strategy against our attacks is to isolate the stack from the regular data memory.

Lockdown [167] is a DBI-based (dynamic binary instrumentation) CFI implementation with a shadow stack. DBI re-writes the binary code of the application at run time, hence, it can control application memory accesses. This allows it to prevent access and leakage of the shadow stack address. However, these security guarantees come with an average run time overhead of 19% which is considered impractical.

Recently LLVM integrated a component of CPI, called SafeStack [118, 210]. It aims to isolate critical stack data, like return addresses or spilled registers from buffers that potentially can be overflowed. During a static analysis phase the compiler identifies buffers that are located on the stack and relocates them to a separate stack, called the *unsafe stack*. The regular stack is then assumed to be the *safe stack*. The separation of buffers and critical values is likely to prevent most stack-based memory vulnerabilities from being exploitable. However, if we can leak the stack pointer register (see Section 3.2.3.2), i.e., the pointer to the *safe stack*, we can overwrite the protected values.

Full CPI [118] and provides more comprehensive protection of code pointers through isolation. On 32-bit x86 the isolation is enforced through segmentation. In principle,

this can prevent our attack attacks, however, on 64-bit x86 or other architectures, e.g., ARM, this feature is not available. The authors suggest alternative implementations to the segmentation-based isolation. All come with their own pros and cons: While the randomization approach provides good performance, it was shown to be prone to information leakage attacks [68]. A more secure implementation is based on software fault isolation (SFI) [225], however, this adds an additional 7% [187] to the 8% average run-time overhead induced by CFI itself [118]. In general, the overhead depends on the number of objects that must be protected, e.g., the authors report of CPI an overhead of 138% for a web server that serves dynamic webpages, which is impractical.

3.2.4.3 *Securing CFI Implementations*

Zeng et al. [236] compiled a list of requirements to implement a secure inline-reference monitor, e.g., for CFI, in which they also mention the danger of stack-spilled variables. However, the threat of stack-spilled registers was not considered in two major compiler implementations. Our work proves that register spills are a severe threat to CFI, which should be address by future implementations.

Ultimately, while stack-oriented defenses help to mitigate stack vulnerabilities, they do not offer sufficient protection to complex software such as web browsers, where dynamic code generation, heap vulnerabilities and attacker-controlled scripting provide many alternative attack vectors to the attacker. Defenders must combine these types of defenses with other protection against heap-based memory corruption to be secure.

3.2.5 *Discussion*

Memory disclosure was previously used to attack code-randomization schemes [196]. Although attacking code randomization is not the main focus of this section, it suggests itself to use stack disclosure against code randomization. In particular, we investigated the impact of stack disclosure against mitigation schemes that aim to prevent direct memory disclosure by marking the code segment as execute-only: XnR [17] and HideM [80]. We performed some preliminary experiments in which we used our capabilities to read the stack of a parallel thread to disclose a large number of return addresses. Considering that we can control which functions are executed in the parallel thread, we were able to leak the addresses of specific gadgets. The results of our experiments are that indirect code disclosure, i.e., disclosing data pointers to infer information about the randomized code layout, [51, 61] via return addresses can be used to bypass fine-grained code-randomization schemes, e.g., function permutation [113] or basic-block permutation [227], that are protected by XnR or HideM. Readactor by Crane et al. [51] performs code-pointer hiding and hence, does not seem vulnerable to return address leakage. Further, the authors extended their work to protect function tables as well [52] which prevents vTable hijacking as described in Section 3.2.3.1.

3.2.6 Conclusion

In this section, we present StackDefiler a set of stack corruption attacks that we use to bypass CFI implementations. Our novel attack techniques corrupt the stack without the need for stack-based vulnerabilities. Doing so we contradict the widely held belief that stack corruption is a solved problem. To the best of our knowledge, this section presents the first comprehensive study of stack-based memory disclosure and possible mitigations.

Surprisingly, we find that fine-grained CFI implementations for the two premier open-source compilers (used to protect browsers), LLVM and GCC, are not safe from attacks against our stack attacks. IFCC spills critical pointers to the stack which we can exploit to bypass CFI checks. We verified that a similar vulnerability exists in VTV—a completely separate implementation of fine-grained CFI in a separate compiler. Next, we demonstrated that unprotected context switches between the user and kernel mode can lead to a bypass of CFI. Further, we show the challenges of implementing a secure and efficient shadow stack and provide evidence that information disclosure poses a severe threat to shadow stacks that are not protected through memory isolation. Finally, we analyzed several stack-based defenses and conclude that none are able to counter our StackDefiler attack.

Based on our findings, we recommend that new defenses should (i) consider the threat of arbitrary memory reads and writes to properly secure a web browser and other attacker-scriptable programs, (ii) never trust values from writable memory, and (iii) recommend complementary approaches to protect the stack and heap to mitigate the threat of memory disclosure.

3.3 FUNCTION-REUSE ATTACKS: ATTACKS ON COARSE-GRAINED CFI

In the previous section, we bypassed (fine-grained) Control-flow Integrity (CFI) by exploiting a compiler-introduced vulnerability. In this section, we will turn our attention to bypassing coarse-grained CFI by exploiting the overly permissive CFI policy. This means that we can change the control flow within the enforced policy boundaries, yet still execute malicious payloads. Our attack, called Counterfeit Object-oriented Programming (COOP), chains C++ virtual functions together to achieve Turing-complete code execution. COOP can successfully bypass real-world coarse-grained CFI implementations that Microsoft [137] and Intel [107] are currently deploying.

3.3.1 *Background on Coarse-grained CFI and C++*

In the following we first provide an explanation of Microsoft's and Intel's CFI implementation, and of the memory representation of C++ objects which are essential for COOP attacks. We then explain the main idea of COOP, and how to increase its resilience against ad-hoc mitigations.

3.3.1.1 *Coarse-grained CFI*

In a recent effort to mitigate code-reuse attacks, Microsoft implemented CFI for Windows 8 and enabled it by default for Windows 10. Their CFI implementation, named Control-flow Guard (CFGuard) [137], enforces coarse-grained CFI for forward branches, and fine-grained CFI for backward branches by implementing a shadow stack. While CFGuard is currently implemented and enforced in software, it was designed during a collaboration with Intel who provides a preview of the planned Instruction Set Architecture (ISA) extension, called Control-flow Enforcement Technology (CET) [107] that can enforce CFGuard in hardware. We categorize CFGuard's CFI policy for forward branches as coarse-grained CFI because indirect jump and call instructions can target *any* benign jump or call target of the entire application during run time. For backward branches, i.e., return instructions, CFGuard leverages a shadow stack to ensure that a function can only return to its caller. As a consequence, traditional [192], as well as sophisticated [61, 196] return-oriented programming (ROP) attacks are mitigated by CFGuard.

Before we show how the attacker can exploit virtual function calls to execute arbitrary malicious payloads without violating CFGuard's policy, we provide the necessary technical background on C++ in the next section.

3.3.1.2 *C++ Virtual Function Calls*

C++ is an extension of the C language to support object-oriented programming. It allows for the declaration of *classes*, which are custom data structures that contain primitive data types such as integers or chars, nested classes, and can be associated with functions. A class is called subclass if it extends another class which is then called parent class.

The relationship is called inheritance, and, as the name suggests, the subclass inherits all properties of the parent class. Another important feature of C++ is the support for polymorphism. In particular, a subclass can override functions of the base class if the function of the parent class was declared a *virtual* function.

Next, we explain with the following example program how these concepts work in practice, and how attackers take advantage of function overriding.

```

1  #include <iostream>
2
3  class Animal {
4      public:
5          virtual void print_animal_name(void) = 0;
6          int get_length(void);
7
8      private:
9          int length;
10 };
11
12 int Animal::get_length(void) {
13     return length;
14 }
15
16 class Dog : public Animal {
17     public:
18         virtual void print_animal_name(void);
19 };
20
21 class Cat : public Animal {
22     public:
23         virtual void print_animal_name(void);
24 };
25
26 void Dog::print_animal_name(void) {
27     std::cout << "Dog" << std::endl;
28 }
29
30 void Cat::print_animal_name(void) {
31     std::cout << "Cat" << std::endl;
32 }
33
34 void print_name(Animal *a) {
35     a->print_animal_name();
36 }
37
38 int main(int argc, char **argv) {
39     if(argc > 2) print_name(new Dog());
40     else print_name(new Cat());
41     return 0;
42 }

```

Listing 7: Example C++ program that demonstrates the concept of virtual functions.

Listing 7 shows the definition of a class `Animal` (Line 3). The `Animal` class defines two functions of which `print_animal_name()` (Line 5) is marked as virtual which means it can be overridden by other classes that inherit from the `Animal` class. In fact, the program declares two further classes, `Dog` (Line 16) and `Cat` (Line 21), which both extend the `Animal` class and override the `print_animal_name()` (Line 26 and Line 30). Depending on the number of arguments that are passed to the `main()` function, it either instantiates a

Dog or a Cat class and passes a pointer to this object as an argument to the `print_name()` function. During compile time it is impossible for the compiler to determine which implementation of `print_animal_name()` should be called (Line 35) because it depends on the number of arguments with which the application is executed (Line 38-40). To resolve the destination of the function call during run time, the compiler generates for each class, which contains virtual functions, a virtual function table (vtable).

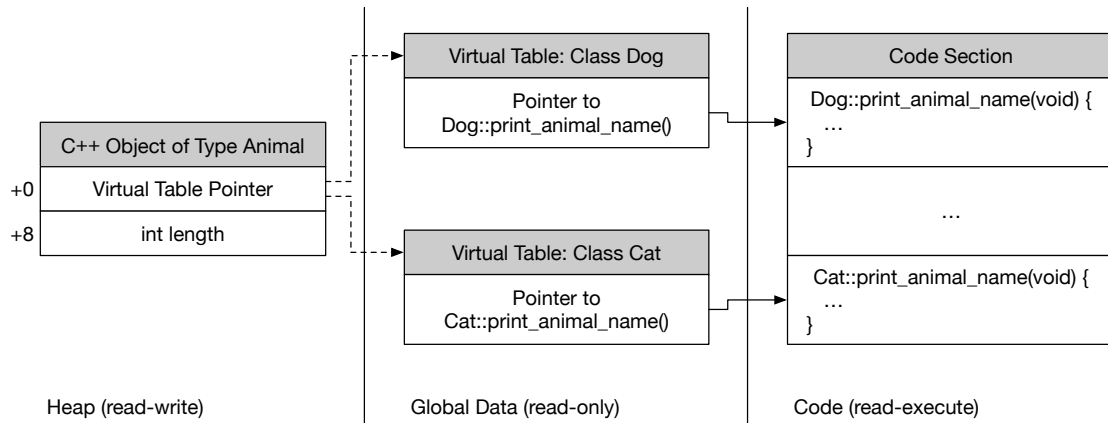


Figure 21: Memory representation of C++ memory objects.

Figure 21 shows the simplified memory representation of the C++ objects that are allocated by the `main()` function in Listing 7. The virtual table (vtable) pointer, which is stored at the beginning of the memory of the object, is set either to the Dog or the Cat vtable. The vtable of a class contains all virtual function pointers. During run time, `print_name()` (Line 34) first dereferences the virtual table, and then the function pointer to resolve the correct destination of the indirect call (Line 35).

From a security perspective, the concept of C++ vtables has two problems: First, the vtable pointer is writable. Attackers can overwrite a vtable pointer of an object with an address to attacker-controlled memory, which contains function pointers to arbitrary addresses, to hijack the control flow. In fact, most of today's exploits against C++ applications rely on overwriting a vtable pointer. Second, if CFI is applied to a C++ application, the CFI implementation needs to be aware of C++ semantics, like virtual tables. If C++ semantics are not considered, the enforced CFI policy must allow every virtual function call site to target any virtual function to avoid breaking the application.

As we demonstrate next, a relaxed policy such as this, as well as coarse-grained CFI, e.g., CFGuard and CET, provides ample freedom for the attacker to chain virtual function calls together to gain arbitrary code execution.

3.3.2 Counterfeit Object-oriented Programming

Counterfeit Object-oriented Programming (COOP), like ROP, is a Turing-complete code-reuse attack [186]. The idea is to exploit a memory-corruption vulnerability to inject counterfeit C++ objects, and then to hijack the control flow to execute a virtual function of each injected object. The main distinction from traditional ROP attacks [192]

are: 1) COOP gadgets consist of whole virtual functions instead of small instruction sequences that end in a return, 2) COOP gadgets are chained through indirect calls instead of return instructions, and 3) COOP overlaps member variables of C++ objects to pass values between two gadgets instead of registers.

Next, we will explain how COOP gadgets are chained, and how values are passed between gadgets.

3.3.2.1 Chaining COOP Gadgets

In COOP, gadgets are connected using a *Main Loop Gadget* (ML-G). The purpose of the ML-G is to iterate over all injected objects, and invoke a virtual function of each object. Therefore, the ML-G is typically comprised of a benign function of the application that contains a loop, which iterates over a list of objects, e.g., an array or linked list of objects. From an application's perspective, ML-G functions are used, e.g., to perform the necessary de-initialization of a list of objects before deleting them.

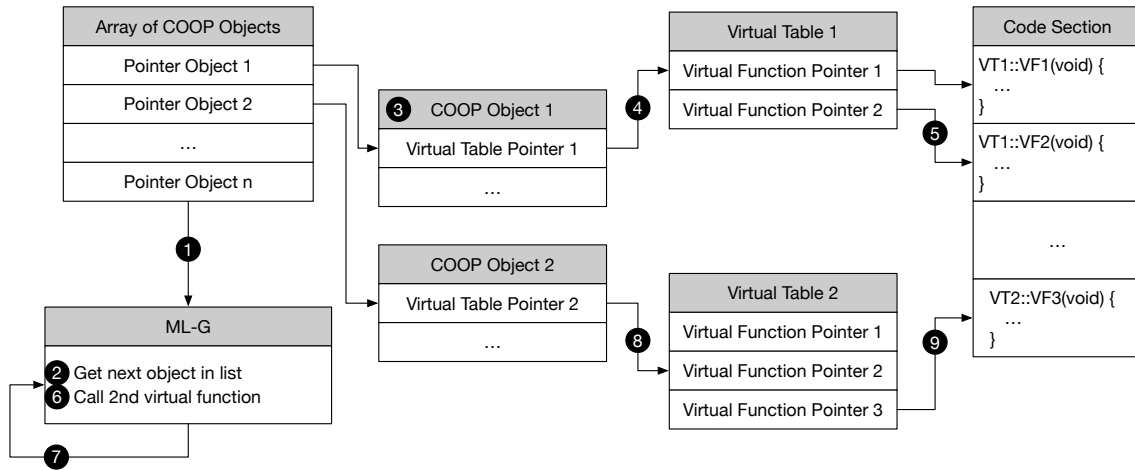


Figure 22: Process of chaining COOP gadgets.

Figure 22 shows in detail how the ML-G works: The attacker hijacks the control flow by exploiting a memory-corruption vulnerability to execute the ML-G, and provides a list of pointers that references injected COOP objects ①. The ML-G ② then takes the first object in the list ③, dereferences the vtable pointer ④, reads the virtual function pointer by adding a constant offset (here plus one entry ⑤), and finally invokes the function ⑥. After the virtual function returns, the ML-G will take the next COOP object from the list, and repeat the previous action ⑦. Note that the ML-G will always call the second virtual function of a referenced vtable and, it is unlikely that all required virtual function gadgets are referenced by the second entry of a vtable. However, the attacker can easily overcome this limitation by setting the vtable pointer to an entry of vtable, instead of the beginning, such that when the ML-G adds the constant offset it results in the desired function. For example, in Figure 22 the attacker crafts the second COOP objects such that the vtable pointer points to the second entry of the vtable ⑧. Hence,

by dereferencing the second entry, the ML-G actually dereferences and calls the third entry ⑨.

3.3.2.2 Unrolled COOP

It is natural to question whether COOP attacks can be mitigated by eliminating potential ML-Gs in an application. To disprove this hypothesis, we developed two refined versions of COOP that do not require ML-Gs and emulate the original *main loop* through *recursion* and *loop unrolling*.

Given a virtual function with not only *one* virtual function invocation but *many*, it is also possible to mount a simple *unrolled* COOP attack that does not rely on a loop or recursion. Consider for example the following virtual function:

```

1 void C::func() {
2     delete obj0;
3     delete obj1;
4     delete obj2;
5     delete obj3;
6 }
```

Listing 8: Example C++ program that demonstrates virtual functions

If objects `obj0` through `obj3` each feature a virtual destructor, `C::func()` can be misused to consecutively invoke four virtual functions. We refer to virtual functions that enable unrolled COOP as UNR-Gs.

We found that even long UNR-Gs are not uncommon in larger C++ applications. For example, in recent Chromium versions, the virtual destructor of the class `SVGPatternElement` is an UNR-G allowing for as many as 13 consecutive virtual function invocations. In practice, much shorter UNR-Gs are already sufficient to compromise a system; we demonstrate in Section 3.3.3 that the execution of three virtual functions is sufficient for the attacker to execute arbitrary code.

3.3.2.3 Data flow between COOP Gadgets

To pass values from one gadget to the next, traditional ROP attacks use registers, however, this is not possible for COOP attacks. This is due to the fact that COOP reuses whole functions which follow calling conventions. For example, when the ML-G invokes a virtual function that loads a value from memory into a callee-save register, this register is restored before the virtual function returns and the ML-G invokes the next function. However, for a successful attack, passing values from one gadget to another is inevitable because each gadget performs only a specific task, e.g., reading or writing a value from memory, or performing an arithmetic operation.

We overcome this restriction by *overlapping* C++ objects to enable data flow between gadgets. Overlapping means that two objects share the same memory, as illustrated in Figure 23. The array of COOP objects ① is the same as in Figure 22. However, the difference is that in Figure 23 both objects share memory. This is possible because the attacker can set the object pointers to arbitrary addresses. In this example, the size of the first object is 32 byte ②. Instead of setting the pointer of the second object to the offset

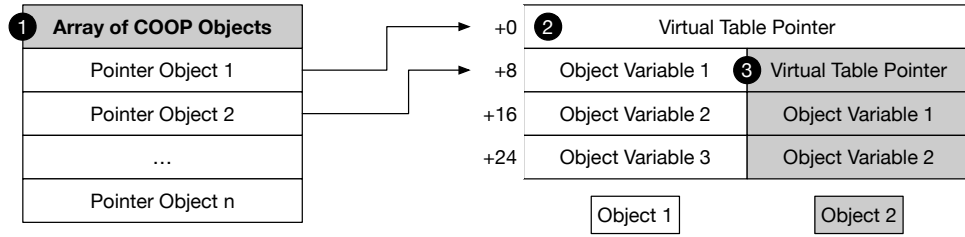


Figure 23: Concept of overlapping C++ objects.

+32 byte, we can set the pointer to point at the offset +8 byte 3. Hence, the memory at offset +16 byte is interpreted as variable 2 or variable 1 depending on if the virtual function of the first or second object is executed. This enables data flow between gadgets, e.g., the first gadget writes a value to variable 2, and the second gadget reads a value from its variable 1 to perform a different operation.

3.3.3 Evaluation

To evaluate the practical strength of COOP, we created attacks against the three most popular browsers (Internet Explorer, Firefox, and Chrome). We then continued our evaluation by analyzing whether COOP attacks could be prevented by existing CFI solutions.

For Chrome, we re-introduced an exploitable bug (CVE-2014-3176) into a recent version of Chromium on Ubuntu 14.04 64-bit. The vulnerability allows an attacker to manipulate JavaScript objects in memory and, consequently, to construct an arbitrary memory read-write primitive. This can be used to reliably disclose memory pointers and hijack the control flow.

We created a COOP exploit for the vulnerability. As is common practice, our exploit changes the protection of a memory region in order to execute injected shellcode. For a successful COOP attack, an attacker must use three virtual function gadgets: The first function loads all needed arguments into the register, and the second calls the memory protection function. The final gadget is an extra function to chain the gadgets. This third function may be a conventional ML-G, or an UNR-G (see Section 3.3.2.2). In our experiments, we successfully executed our attack on an unprotected version of Chromium.

After confirming that COOP is as powerful as state-of-the-art ROP attacks [196], we turn our attention to its effectiveness in the presence of CFI. We found that coarse-grained CFI, as it is enforced by CFGuard, is ineffective against COOP attacks because they do not violate the policy. We further found that CFI solutions, which operate only on the binary [5, 75, 143, 173, 237, 238], and those which do not consider C++ semantics [11, 152, 215] are ineffective as well.

COOP can be mitigated by making CFI C++ aware [111], or by our novel randomization-based defense, which we explain in more detail in Section 4.1.2.

3.3.4 *Conclusion*

In this section, we introduced a novel code-reuse attack technique, called Counterfeit Object-oriented Programming (COOP), which attackers can exploit to bypass coarse-grained CFI. COOP chains C++ virtual functions through the main-loop gadget together to achieve arbitrary code execution. We further demonstrate that in practice the attacker does not rely on the existence of a main-loop gadget but can leverage the destructor of C++ objects.

3.4 DOJITA: DATA-ONLY ATTACK ON JIT COMPILERS

Dynamic programming languages, like JavaScript, are increasingly popular since they provide a rich set of features and are easy to use. They are often embedded into other applications to provide an interactive interface. Web browsers are the most prevalent applications embedding JavaScript run-time environments to enable website creators to dynamically change the content of the current web page without requesting a new website from the web server. For efficient execution, modern run-time environments include just-in-time (JIT) compilers to compile JavaScript programs into native code.

Code-injection/reuse. Unfortunately, the run-time environment and the application that embeds dynamic languages often suffer from memory-corruption vulnerabilities due to the usage of unsafe languages such as C and C++ that are still popular for compatibility and performance reasons. Attackers exploit memory-corruption vulnerabilities to access memory (unintended by the programmer), corrupt code and data structures, and take control of the targeted software to perform arbitrary malicious actions. Typically, attackers corrupt code pointers to hijack the control flow of the code, and to conduct *code-injection* [9] or *code-reuse* [151] attacks.

While code injection attacks have become less appealing, mainly due to the introduction of Data Execution Prevention (DEP) or writable xor executable memory ($W\oplus X$), state-of-the-art attacks deploy increasingly sophisticated code-reuse exploitation techniques to inject malicious code-pointers (instead of malicious code), and chain together existing instruction sequences (*gadgets*) to build the attack payload [192].

Code-reuse attacks are challenging to mitigate in general because it is hard to distinguish whether the execution of existing code is benign or controlled by the attacker. Consequently, there exists a large body of literature proposing various defenses against code-reuse attacks. Prominent approaches in this context are code randomization and control-flow integrity (CFI). The goal of code randomization [119] schemes is to prevent the attacker from learning addresses of any gadgets. However, randomization techniques require extensions [17, 27, 51, 52, 80] to prevent information-disclosure attacks [61, 190, 196]. Control-flow integrity (CFI) [5] approaches verify whether destination addresses of indirect branches comply to a pre-defined security policy at run time. Previous work demonstrated that imprecise CFI policies in fact leave the system vulnerable to code-reuse attacks [31, 32, 45, 60, 83, 84, 186]. Further, defining a sufficiently accurate policy for CFI was shown to be challenging [69].

Data-only attacks. In addition to the aforementioned attack classes, *data-only* attacks [38] have been recently shown to pose a serious threat to modern software security [99]. Protecting against data-only attacks in general is even harder because any defense mechanism requires the exact knowledge of the input data and the intended data flow. As such, solutions that provide memory safety [148, 149] or data-flow integrity [33] generate impractical performance overhead of more than 100%.

JIT attacks. Existing defenses against the attack techniques mentioned above are mainly tailored towards static code making their adoption for dynamic languages difficult. For example, the JIT-compiler regularly modifies the generated native code at run time for optimization purposes. On the one hand, this requires the code to be writable, and hence,

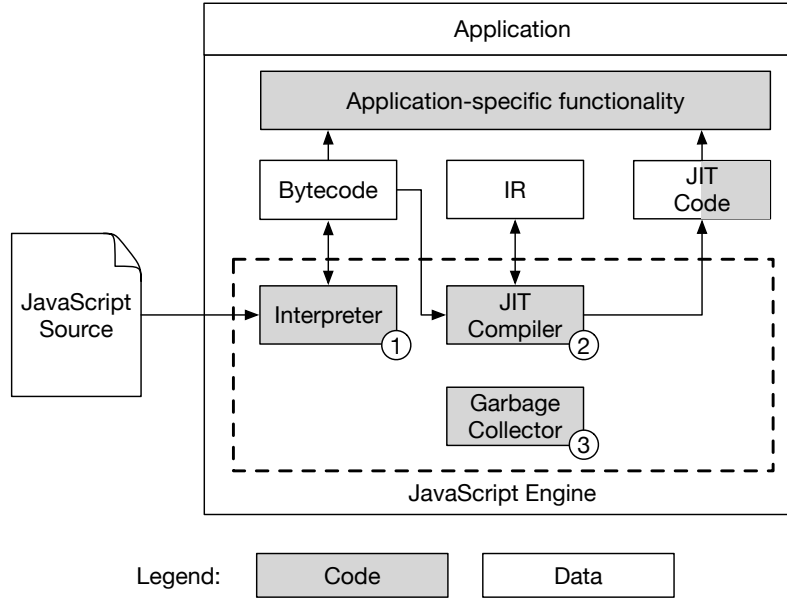


Figure 24: Main components of a JavaScript JIT engine.

enables code-injection attacks. On the other hand, it makes state-of-the-art defenses challenging to adopt, either due to the increased performance overhead in the case of CFI [154] (+9.6%; in total 14.6%)², or due to unclear practicability of code-pointer hiding [51] to protect code-randomization applied to the JIT code. Further, the attacker controls the input of the JIT compiler, and can input a program that is compiled to native code containing all required gadgets. Finally, the attacker can tamper with the input of the JIT compiler to generate malicious code, as we show in Section 3.4.3.

Goals and Contributions. In this section, we present a novel data-only attack against the JIT compiler that allows to execute arbitrary code, and can bypass all existing code-injection and code-reuse defenses. Concurrently to our work, researchers published a data-only attack that targets internal data structures of Microsoft’s JIT Engine [213].

3.4.1 Background on SGX and JIT Compilation

To understand We start with a short introduction of Intel’s Software Guard Extensions (SGX) [104] which constitutes the trusted computing base for our defense tool JITGuard. Then we explain the basic principles of just-in-time compilers for browsers, which is the main use-case for our proof-of-concept implementation in this chapter.

3.4.1.1 JIT Engines

JIT engines provide a run-time environment for high-level scripting languages, allowing the script to interact with application-specific functionality. They leverage so-called

² Compared to MCFI [153], a CFI implementation by the same author for static code.

just-in-time (JIT) compilers to transform an interpreted program or script into native code at run time. Browsers in particular make heavy use of JIT compilers to increase the performance of *JavaScript* programs. JavaScript is a high-level scripting language explicitly designed for browsers to dynamically change the content of a website, e.g., in reaction to user input. In general, JIT engines consist of at least three main components, as shown in Figure 24: ① an interpreter, ② a JIT compiler and ③ a garbage collector.

The purpose of JIT compilers is to increase the execution performance of JavaScript by compiling the script to native code. Since compilation can be costly, usually not all of the scripting code is compiled. Instead, JIT engines include an interpreter which transforms the input program into not optimized *bytecode*, which is executed by the interpreter. During the execution of the bytecode, the interpreter profiles the JavaScript program to identify parts (i.e., functions) of the code which are executed frequently (*hot code*). When the interpreter identifies a hot code path, it estimates if compilation to native code would be more efficient than continuing to interpret the bytecode. If this is the case, it passes the hot code to the JIT compiler.

The JIT compiler takes the bytecode as input and outputs corresponding native machine code. Similar to regular compilers, the JIT compiler first transforms the bytecode into an *intermediate representation* (IR) of the program, which is then compiled into native code, also called *JIT code*. In contrast to the bytecode, which is interpreted in a restricted environment through a virtual machine, this native code is executed directly by the processor that runs the browser application. To ensure that malicious JavaScript programs cannot harm the machine of the user, the JIT compiler limits the capabilities of the emitted JIT code. In particular, the compiled program cannot access arbitrary memory, and the compiler does not emit potentially dangerous instructions, e.g., system call instructions. Further, the emitted native code is continuously optimized, and eventually, de-optimized when the JIT compiler determines that this is not needed anymore. Because the JIT compiler has to write the emitted native code to memory as part of its output, the permissions of *JIT code pages* are usually set to read-write-executable.

The last major component is the garbage collector. In contrast to C and C++, in JavaScript the memory is managed automatically. This means that the garbage collector tracks memory allocations and releases unused memory when it is no longer needed.

3.4.1.2 JIT-based Attacks and Defenses

Typically attacks on JIT compilers exploit the read-write-executable JIT memory in combination with the fact that attackers can influence the output of the JIT compiler by providing a specially crafted input program. In the popular pwn2own exploiting contest, Gong [86] injected a malicious payload into the JIT memory to gain arbitrary code execution in the Chrome browser without resorting to code-reuse attacks like return-oriented programming (ROP) [192]. To prevent code-injection attacks, $W \oplus X$ was adapted for JIT code [36, 37, 51, 144]. However, as discussed in the previous section, JIT code pages must be changed to writable for a short time when the JIT compiler emits new code, or optimizes the existing JIT code. Song et al. [199] demonstrated that this small time window can be exploited by attackers to inject a malicious payload. They

JavaScript		
<pre>function foo() { var y = 0x3C909090 ^ 0x90909090; }</pre>		
Native Code		
Address	Opcodes	Disassembly
0:	B8 9090903C	mov eax, 0x3C909090
5:	35 90909090	xor eax, 0x90909090
Unaligned Native Code		
Address	Opcodes	Disassembly
1:	90	nop
2:	90	nop
3:	90	nop
4:	3C35	cmp al, 35
6:	90	nop
7:	90	nop
8:	90	nop
9:	90	nop

Figure 25: During JIT spraying the attacker exploits that large constants are directly transferred into the native code. By jumping into the middle of an instruction the attacker can execute arbitrary instructions that are encoded into large constants.

propose to mitigate this race condition by splitting the JIT engine into two different processes: an untrusted process which executes the JIT code, and a trusted process which emits the JIT code. Their architecture prevents the JIT memory from being writable in the untrusted process at any point in time. Since the split JIT engine now requires inter-process communication and synchronization between the two processes, the generated run-time overhead can be as high as 50% for JavaScript benchmarks. Further, this approach does not prevent code-reuse attacks. Microsoft [141] recently adapted out-of-process JIT generation for their JavaScript engine Chakra to avoid remapping the JIT-code region as writable during code generation. This is done by using a double mapping of the JIT-code region which is mapped as read-execute in the untrusted and read-write in the trusted process. The compiler executes in the untrusted process, generates the JIT code and sends it to the trusted process, which then copies it to the double-mapped region and signals to the untrusted process that it is ready for execution.

Code-reuse attacks chain existing pieces of code together to execute arbitrary malicious code. JIT engines facilitate code-reuse attacks because the attacker can provide input programs to the JIT compiler, and hence, influence the generated code to a certain degree. However, as mentioned in Section 3.4.1.1, the attacker cannot force the JIT compiler to emit arbitrary instructions, e.g., system call instructions which are required for most exploits. To bypass this restriction Blazakis [24] observed that numeric constants in a JavaScript program are copied to the JIT code, as illustrated in Figure 25: the attacker can define a JavaScript program which assigns large constants to a variable, here the result of $0x3C909090 \text{ xor } 0x90909090$ is assigned to the variable *y*. When the compiler transforms this expression into native code, the two constants are copied into

the generated instructions. This attack is known as *JIT spraying* and enables the attacker to inject 3-4 arbitrary bytes into the JIT code. By forcing the control flow to the middle of the `mov` instruction, the CPU will treat the injected constant bytes as an instruction and execute them.

JIT spraying can be mitigated by constant blinding, i.e., masking large constant C through xor with a random value R at compile time. The JIT compiler then emits an xor instruction to unblind the masked constant before using it ($((C \oplus R) \oplus R = C \oplus 0 = C)$). While constant blinding indeed prevents JIT spraying it decreases the performance of the JIT code. Further, Athanasakis et al. [13] demonstrated that JIT spraying can also be performed with smaller constants, and that constant blinding for smaller constants is impractical due to the imposed run-time overhead. Recently, Maisuradze et al. [127] demonstrated a JIT-spraying attack by controlling the offsets of relative branch instructions to inject arbitrary bytes into the JIT code.

Another approach to mitigate JIT-spraying is code randomization. Homescu et al. [94] adopted fine-grained randomization for JIT code. However, similar to static code, code randomization for JIT code is vulnerable to information-disclosure attacks [196]. While Crane et al. [51] argued that leakage resilience based on execute-only memory can be applied to JIT code as well, they do not implement code-pointer hiding for the JIT code which makes the performance impact hard to estimate. Tang et al. [206] and Werner et al. [229] proposed to prevent information-disclosure attacks through destructive code reads. Their approach is based on the assumption that benign code will never read from the code section. Destructive code reads intercept read operations to the code section, and overwrite every read instruction with random data. Hence, all memory leaked by the attacker is replaced by random data, rendering it unusable for code-reuse attacks. However, Snow et al. [197] demonstrated that this mitigation is ineffective in the setting of JIT code. In particular, the attacker can use the JIT compiler to generate multiple versions of the same code by providing a JavaScript program with duplicated functions. Upon reading the code section the native code of the first function will be overwritten while the other functions are intact and can be used by the attacker to conduct a code-reuse attack.

Niu et al. [154] applied CFI to JIT code and found that it generates on average 14.4% run-time overhead and does not protect against data-only attacks which do not tamper with the control flow but manipulate the data flow to induce malicious behavior.

3.4.2 Threat Model

The main goal is to demonstrate that data-only attacks against the Just-in-Time (JIT) compiler constitute a severe threat. Therefore, our threat model and assumptions exclude attacks on the static code. Our threat model is consistent with the related work in this area [24, 51, 127, 154, 199].

Defense Capabilities

Static code is protected. We assume state-of-the-art defenses against code-injection and code-reuse attacks for static code are in-place. In particular, this means

that code-injection is prevented by enforcing DEP [136], and code-reuse attacks are defeated by randomization-based solutions [51, 52], or (hardware-assisted) control-flow integrity [5, 107, 215]. Additionally, we assume that the static code of the application and the operating system are not malicious.

Data randomization. We assume the targeted application to employ Address Space Layout Randomization (ASLR) [165]. This prevents the attacker from knowing any addresses of allocated data regions a priori. This also enables us to hide sensitive data from the attacker.

Adversary Capabilities

Memory-corruption vulnerability. The target program suffers from at least one memory-corruption vulnerability. The attacker can exploit this vulnerability to disclose and manipulate data memory of *known* addresses. This is a common assumption for browser exploits [45, 186, 196].

Scripting Engine. The attacker can utilize the scripting engine to perform arbitrary (sandboxed) computations at run time, e.g., adjust the malicious payload based on disclosed information.

We note that any form of side-channel, e.g., cache and timing attacks to leak randomized memory addresses, or hardware attacks are out of scope.

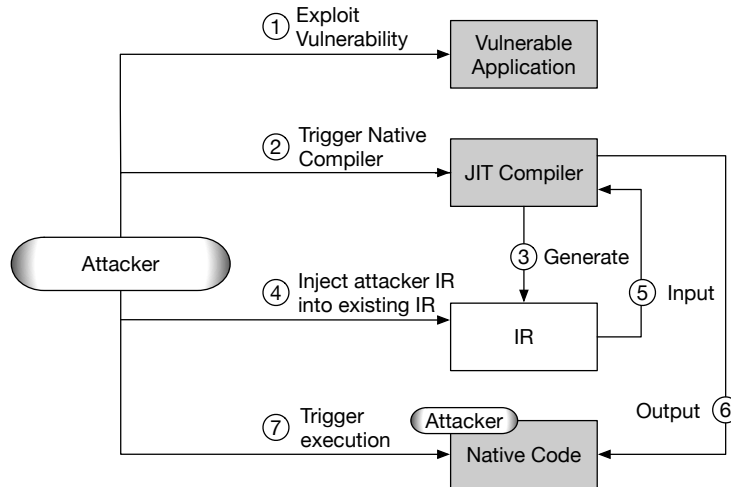


Figure 26: DOJITA enables the attacker to execute arbitrary code through a data-only attack. In particular, the attacker manipulates the IR which is then used by the JIT compiler to generate native code that includes a malicious payload.

3.4.3 Our Data-only Attacks on JIT Compilers

As mentioned in the previous section, existing JIT protections only aim to prevent code-injection or code-reuse attacks. However, in our preliminary experiments we

observed that arbitrary remote code execution is feasible by means of *data-only* attacks which corrupt the memory *without* requiring to corrupt any code pointers. We implemented an experimental data-only attack against JIT compilers, coined DOJITA (Data-Only JIT Attack), that manipulates the intermediate representation (IR) to trick the JIT compiler into generating arbitrary malicious payloads. Our experiments underline the significance of data-only attacks, in the presence of defenses against control-flow hijacking, and will motivate new defenses. Figure 26 shows the high-level idea of DOJITA:

The attacker ① exploits a memory-corruption vulnerability to read and write arbitrary data memory; ② identifies a hot function F in the input program, which will be compiled to native code; ③ during the compilation of F the JIT compiler will generate the corresponding IR; the attacker discloses the memory address of the IR in memory which is commonly composed of C++ objects; ④ injects crafted C++ objects (the malicious payload) into the existing IR. ⑤ Finally the JIT compiler uses the IR to generate the native code ⑥. Since the IR was derived from the trusted bytecode input, the JIT compiler does not check the generated code again. ⑦ Thus, the generated native code now contains a malicious payload and is executed upon subsequent invocations of the function F .

3.4.3.1 Details

For our experiments, we chose to attack the open source version of Edge’s JavaScript engine, called ChakraCore [139]. Our goal is to achieve arbitrary code execution by exploiting a memory-corruption vulnerability without manipulating the JIT code or any code pointers. Further, we assume that the static code and the JIT code are protected against code-reuse and code-injection attacks, e.g., by either fine-grained code randomization [51], or fine-grained (possibly hardware-supported) control-flow integrity [107, 154]).

For our attack against ChakraCore we carefully analyzed how the JIT compiler translates the JavaScript program into native code. We found that the IR of ChakraCore is comprised of a linked list of `IR::Instr` C++ objects where each C++ object embeds all information, required by the JIT compiler, to generate a native instruction or an instruction block. These objects contain variables like `m_opcode` to specify the operation, and variables `m_dst`, `m_src1`, and `m_src2` to specify the operands for the operation. To achieve arbitrary code execution, we carefully craft our own objects, and link them together. Figure 27 shows the IR after we injected our own `IR::Instr` objects (lower part of the figure), by overwriting the `m_next` data pointer of the benign `IR::Instr` objects (upper part of the figure). When the JIT compiler uses the linked list to generate the native code it will include our malicious payload. It is noteworthy that `m_opcode` cannot specify arbitrary operations but is limited to a subset of instructions like (un-)conditional branches, memory reads/write, logic, and arithmetic instructions. This allows us to generate payloads to perform arbitrary computations, and to read and write memory. However, for a meaningful attack we have to interact with the system through system calls. We could inject a `call` instruction to the system call wrapper function which is provided by system libraries. However, this would require leaking the address of the wrapper which might not be possible, e.g., if defenses such as Readactor [51]

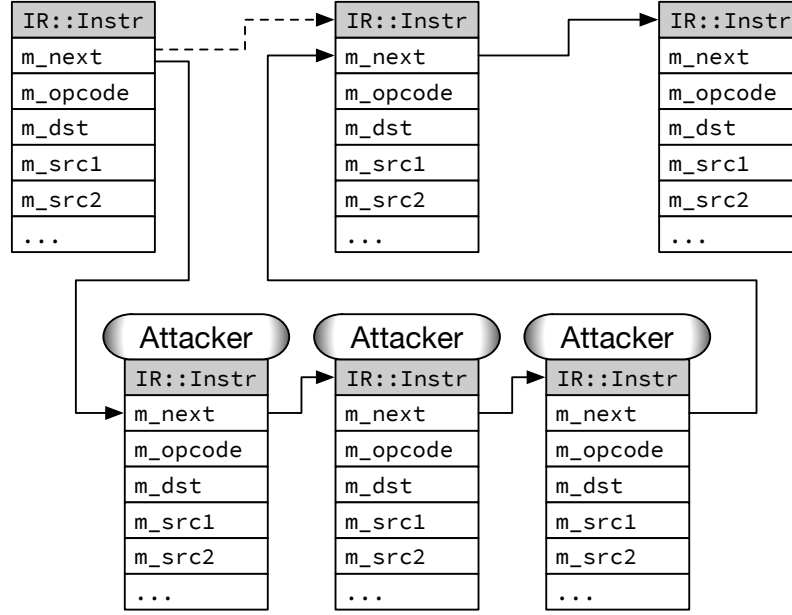


Figure 27: The IR of ChakraCore consists of a linked list of `IR::Instr` C++ objects. The attacker injects instructions by overwriting the `m_next` pointer of a benign object (dotted line) to point to a linked list of crafted objects.

are in place that mitigate the disclosure of code pointers. Hence, we use unaligned instructions [192] by embedding the system call instruction in another instruction. In particular, we could generate an `add` or `jmp` [128] instruction where the operator is set to the constant `0xC3050F` which encodes the instructions `syscall; ret`. Finally, we generate a call instruction into the middle of the `add` instruction to execute the unaligned code and issue a system call.

3.4.3.2 Implementation

For our proof-of-concept of DOJITA we implemented an attack framework that allows the attacker to specify an arbitrary attack payload. Our framework parses and compiles the attack payload to the ChakraCore IR, i.e., the framework automatically generates C++ memory objects that correspond to the instruction of the attack payload. Next, the framework exploits a heap overflow in `Array.map()` (CVE-2016-7190), which we re-introduced to the most recent public version of ChakraCore (version 1.4), to acquire read/write primitive. After disclosing the internal data-structures of the JIT compiler, we modify data pointers within these structures to include our malicious IR. The JIT compiler will then iterate through the IR memory objects, and generate native code. While the injection of malicious IR into the benign IR depends on a race condition we found that the attack framework can reliably win this race by triggering the execution of the JIT compiler repeatedly. In our testing, DOJITA succeeded 99% of the times.

Our proposed data-only attack against the JIT compiler cannot be mitigated by any state-of-the-art defenses or defenses proposed in the literature [51, 154]. The reason is that these defenses cannot distinguish the benign IR from the injected IR.

3.4.3.3 *Comparison to Related Work*

Independently from our work, Theori [213] published a similar attack that also targets the internal data structures of Microsoft’s JIT compiler. Their attack targets a temporary buffer which is used by the JIT compiler during compilation to emit the JIT code. This temporary buffer is marked as readable and writable. However, once the JIT compiler generated all instruction from the IR, it relocates the content of the temporary buffer into the JIT memory which is marked as readable and executable. By injecting new instructions into this temporary buffer, one can inject arbitrary code into the JIT memory. Microsoft patched the JIT compiler to include a cyclic redundancy checksum of the emitted instructions during compilation. The JIT code is only executed if the checksum of the relocated buffer corresponds to the original checksum.

This defense mechanism which was recently added by Microsoft *does not* prevent our attack. While the attack by Theori [213] is similar to ours, we inject our malicious payload at an earlier stage of the compilation. As a consequence, the checksum, which is computed during compilation, will be computed over our injected IR. Since we do not perform any modifications in later stages, the checksum of the relocated buffer is still valid and the JIT compiler cannot detect our attack.

3.4.4 *Conclusion*

Protection of modern software against run-time attacks (code injection and code reuse) has been a subject of intense research and a number of solutions have been deployed or proposed. Moreover, recently, researchers demonstrated the threat of the so-called data-only attacks that manipulate data flows instead of the control flow of the code. These attacks seem to be very hard to prevent because any defense mechanism requires the exact knowledge of the input data and the intended data flow. However, on the one hand, most of the proposed defenses are tailored towards statically generated code and their adaption to dynamic code comes with the price of security or performance penalties. On the other hand, many widespread applications, like browsers and document viewers, embed just-in-time compilers to generate dynamic code.

We present a novel data-only attack, dubbed DOJITA, against JIT compilers that can successfully execute malicious code even in the presence of defenses against control-flow hijacking attacks such as Control-Flow Integrity (CFI) or randomization-based defenses. Specifically, this attack manipulates the intermediate representation of JIT programs, which is used by the compiler to generate the dynamic code, to trick the JIT compiler into generating malicious code. We found that state-of-the-art JIT code defenses cannot mitigate this attack. To protect against DOJITA the internal data structures must be protected against modifications. This is a non-trivial challenge, e.g., isolating the JIT compiler by swapping it out to a separate process is likely to result in a large

performance overhead because the JIT compiler is often invoked during the execution of (compiled) JavaScript for further optimization purposes.

Recently, Frassetto et al. [72] proposed a new design for JIT engines which leverages Intel's Software Guard Extensions (SGX) [104] to isolate the JIT compiler, and hence, mitigates attacks like DOJITA.

3.5 RELATED WORK

In this section, we provide an overview of three classes of related memory-corruption attacks. First, we discuss different techniques of attack mitigations that rely on memory secrecy. We then elaborate on different ways to attack Control-flow Integrity, and conclude with the related work on data-only attacks.

Many exploit mitigations introduce randomness into the in-memory representation of applications. Such mitigations rely on the assumption that the attacker cannot read the memory. However, in the presence of memory-disclosure vulnerabilities, assuming memory secrecy is neither justified nor realistic.

Bhatkar et al. [20] note that contemporary schemes (ASLR, StackGuard [48], PointGuard [49]) are vulnerable if the attacker can read arbitrary values in memory. Strackx et al. [203] later demonstrate that memory disclosure through buffer overread errors allows attackers to bypass ASLR and stack canaries. Fresi Roglia et al. [74] then use return-oriented programming to disclose the randomized location of `libc`. Based on the insight that ASLR was highly vulnerable to simple memory-disclosure attacks, researchers argued that fine-grained code randomization solutions would provide sufficient resilience [59, 82, 93, 109, 119, 162, 227]. However, as we have shown in Section 3.1, the attacker can bypass all randomization-based defenses by leveraging a direct or an indirect memory-disclosure attack.

Bittau et al. [22] develop another memory disclosure attack against services that automatically restart after crashes. This attack exploits the fact that some servers (created using `fork` without `execve`) do not re-randomize after a crash. By sending such servers a malformed series of requests and by analyzing whether the requests cause the server to crash, hang, or respond, the attacker can guess the locations of the gadgets required to launch a simple ROP attack that sends the program binary to the remote attacker. Like Just-in-Time Return-oriented Programming (JIT-ROP) (cf. Section 3.1.2), this attack undermines fine-grained code randomization.

Siebert et al. [188] present a memory disclosure attack against servers that uses a timing side-channel. By sending a malformed request to a web server, the attacker can control a byte pointer that controls the iteration count of a loop. This creates a correlation between the target of the pointer and the response time of the request that the attacker can use to (slowly) scan and disclose the memory layout of the victim process. In a similar vein, Hund et al. [101], Wojtczuk [232], Jang et al. [112], and Gruss et al. [89] exploit a timing side-channel to infer the memory of the privileged ASLR-randomized kernel address space.

Gras et al. [88] present an evict and time cache side-channel attack against Address Space Layout Randomization (ASLR) that can be launched from within the JavaScript sandbox. Hence, attackers can use this technique when executing an attack against browsers to bypass ASLR without relying on an information-disclosure vulnerability.

Evans et al. [68] use a memory-disclosure attack to bypass an implementation of the code pointer integrity (CPI) by Kuznetsov et al. [118]. CPI works by storing control flow and bounds information in a safe region which is separate from non-sensitive data. This prevents control-flow hijacking and spatial memory corruption. Whereas the

32-bit x86 implementation uses memory segmentation to isolate the safe region, the fastest 64-bit x86 implementation uses randomization to implement information hiding. However, it turns out that the hidden safe region was sufficiently large to be located and parsed using a modified version of the memory disclosure attack by Siebert et al. [188]. Kuznetsov et al. [118] also provide a 64-bit CPI implementation where the safe region is protected by Software Fault Isolation (SFI), which has not been bypassed.

Göktaş et al. [85] and Oikonomopoulos et al. [157] discuss attack techniques to break memory secrecy by lowering the entropy. Specifically, many modern client and server applications provide interfaces that the attacker can use to reliably perform memory allocations. The attacker can exploit allocation oracles to reliably guess the address of *hidden* memory.

Gawlik et al. [76] present *Crash Resistant Oriented Programming* which enables brute-force attacks against randomization-based defenses. Their attack combines information-disclosure vulnerabilities with fault-tolerant functionality of browsers to probe memory addresses. In particular, they found that threads, which the attacker can spawn through JavaScript, install their own exception handler, which prevents the browser from crashing if a thread accesses an invalid memory address. In a follow-up work Kollenda et al. [116] explore techniques which attackers use to automate the process of finding crash-resistant primitives.

Snow et al. [197] present memory-disclosure attacks against defenses that aim to prevent direct memory-disclosure of randomized code by means of *destructive code reads* [206, 229]. This means that every byte of the code section is overwritten with a random byte after it was read. However, this does not prevent an attacker from performing direct disclosure attacks. In particular, the attacker can exploit the Just-in-Time (JIT) engine of a browser or document viewer to generate native code of two identical functions. The attacker then discloses the content of the first function to find the addresses of suitable code-reuse gadgets. Since the two functions were identical, the attacker can use the gadgets of the second function for her attack. Further, the attacker can in some cases reload a shared library after disclosing its content. Finally, Snow et al. [197] find that the attacker can guess part of the randomized code. For example, a function epilog performs the reversed operations of a function prolog, hence, by disclosing the function prolog the attacker can reliably guess the instruction of the function epilog. This attack assumes the usage of certain code-randomization schemes. Pewny et al. [171] demonstrate that the combination of code inference and whole-function reuse can bypass destructive code reads, regardless of the applied code-randomization technique.

Rudd et al. [179] explore the possibilities of reusing code pointers, which are protected by a layer of indirection, e.g., by means of code pointer hiding (cf. Section 4.1.2.3). Their results show that, in the case of a vulnerable web server, pointer protection through a layer of indirection is not sufficient and gives the attacker enough leeway to gain arbitrary code execution. van der Veen et al. [222] present a generalization of this idea in the form of an analysis framework which identifies the remaining attack surface for an application after a certain mitigation has been applied. Specifically, it allows the definition of a number of constraints which are imposed on the attacker by a code-reuse

mitigation. The result of the analyses are gadgets which do not violate these constraints, and hence, can be leveraged for a code-reuse attack.

3.5.1 *Attacks against Control-flow Integrity*

Control-flow Integrity is one of the promising alternatives to randomization-based defenses that are effective in mitigating code-reuse attacks. However, as for randomization-based defenses, researchers identified pitfalls of different Control-flow Integrity (CFI) schemes as well.

After Shacham [192] published his work on return-oriented programming (ROP), researchers focused on ensuring the integrity of return addresses on the stack [5, 56, 57]. Checkoway et al. [35] and Bletsch et al. [25] present an alternative technique of performing ROP attacks, coined Jump-oriented Programming (JOP). The principle of JOP is the same as ROP: the attacker chains short instruction sequences together in order to execute arbitrary malicious payloads. However, instead of leveraging return instruction to chain gadgets, JOP leverages on indirect jump instructions. To chain gadgets through an indirect jump instruction, JOP either relies on gadgets that end in `pop reg / jmp reg` pair, and hence, emulate a return instruction, or on a dispatcher gadget. A dispatcher gadget can chain JOP gadgets that all end with `jmp reg` by first setting `reg` to the start address of the dispatcher gadget, then loading the next JOP gadget pointer from an attacker provided list of JOP gadget pointers into a register, and finally jumping to it.

Lettner et al. [121] port the idea of Counterfeit Object-oriented Programming (COOP) to Objective-C. Specifically, they exploit the `msgSend()` dispatcher function which makes this attack hard to defeat. Their attack shows that COOP style attacks are practical and a powerful alternative to classic ROP attacks.

Zhang and Sekar [239] propose binCFI which is similar to the original CFI by Abadi et al. [5] but relaxes the CFI policies to improve run-time performance. The enforced CFI policy by binCFI for return instruction requires return addresses to target a call-preceded instruction. However, Davi et al. [60] and Göktaş et al. [83] demonstrate that this policy is too imprecise for real-world applications. In particular, they found that the attacker can construct Turing-complete ROP chains that conform to the CFI policy of binCFI.

Pappas et al. [163], Cheng et al. [43], and Fratric [73] present CFI schemes that aim to mitigate ROP attacks, and to have a better performance than traditional shadow stacks [56]. The main idea of these schemes is to only verify the return address(es) on the stack when an attack is likely to be executed, e.g., during a system call because most attack payloads need to interact with the kernel at some point to perform malicious actions. For efficiency reasons these schemes only verify a configurable amount of return addresses with the help of heuristics to detect ROP attacks, e.g., if a certain number of return addresses point to short instruction sequences. Carlini and Wagner [31] Davi et al. [60], Schuster et al. [185], and Göktaş et al. [84] present attacks that demonstrate that these heuristics provide the attacker with enough freedom to perform Turing-complete attacks. Wollgast et al. [233] implement a gadget finding framework, which discovers

ROP gadgets that conform to a given CFI policy. In particular, their attacks demonstrate that real-world applications contain ROP gadgets, which are indistinguishable from benign instruction sequences that end in a return instruction. Hence, they can be exploited by the attacker to evade the heuristics that aim to detect traditional ROP gadgets. Further, they demonstrate the limitations of Central Processing Unit (CPU) features like the Last Branch Record (LBR) for enforcing CFI as the attacker can flush the LBR to evade the detection of ROP payloads.

Carlini et al. [32] explore the limitations of static fine-grained CFI that does not consider any state information. This means that a return instruction cannot only return to the original caller of its function but to any potential caller. With their attack, coined control-flow bending, Carlini et al. [32] provide evidence that a shadow stack is required to provide precise enforcement of the control-flow graph (CFG).

In general, the effectiveness of CFI greatly depends on the precision of the enforced CFG that is derived for the protected application. The aforementioned attacks on CFI mainly exploit the fact that the attacked CFI schemes decrease the precision of the enforced CFG on purpose to lower the performance overhead. Evans et al. [69] assume the best-case scenario (from a defender's perspective) to create the CFG, which provides access to the source code and compile-time information. In Control Jujutsu they analyze the precision of the state-of-the-art algorithm [120] that is used to derive forward edges in the CFG, and find that the derived CFG contains imprecisions that can be exploited and thus allow arbitrary code execution.

3.5.2 *Data-only Attacks*

Data-only attacks are an attack class which, unlike code-injection and code-reuse attacks, does not require the attacker to maliciously modify the control-flow graph. As a consequence, previously discussed defenses that attempt to mitigate control-flow attacks are ineffective in terms of preventing data-only attacks.

Chen et al. [38] demonstrate data-only attacks against server applications. Specifically, they show how the attacker can manipulate the data flow within the application to bypass authentication check, disclose configuration files, and even escalate privileges by changing decision-making data.

Heartbleed [133] is a buffer-overread bug in OpenSSL [211] which allows the attacker to reliably disclose memory of server applications that utilize OpenSSL. The Heartbleed bug affected a wide variety of applications and allowed in most cases for the attacker to execute a data-only attack to disclose the private keys of the server or data of other users.

Hu et al. [98] present FlowStitch which tries to automate the process of creating data-only attacks. The idea is that the attacker provides three inputs to the FlowStitch framework which then automatically generates a data-only attack: the first input triggers a memory-corruption vulnerability in the target application, the second one triggers the execution of the same execution path as the first input without crashing the application, and the third input identifies data that are interesting to the attacker, e.g., cryptographic

keys. FlowStitch then automatically generates an exploit that combines different data flows within the application that allow the attacker to exfiltrate the interesting data.

In a follow-up, Hu et al. [99] extend their previous work by introducing the notion of Data-oriented Programming (DOP). Their main goal was to subvert existing CFI defenses by showing how data-only attacks can be used to implement Turing-complete attacks without changing the control-flow.

Browsers pose an interesting target because they are highly complex software and serve as an execution environment for web applications. As a result, browsers handle some of the most sensitive data that range from banking information, credit card numbers and passwords. Rogowski et al. [178] explore data-only attacks in the context of a browser. Specifically, they present a framework that automatically generates exploits that perform a data-only attack to disclose authentication tokens, or enable malicious websites to bypass the same-origin policy which normally isolates different websites from one other.

3.6 SUMMARY AND CONCLUSION

In this chapter, we demonstrated novel memory-corruption attacks against code randomization and control-flow integrity.

In the first part of this chapter we presented novel information-disclosure attacks. Our attacks repeatedly leverage information-disclosure vulnerabilities without crashing the application. We use this primitive to disclose the content of the code section to analyzing the randomized code, and to compile a customized return-oriented programming payload during run time. We further show that even if the attacker cannot directly read the code but is limited to the disclosing content of the data section she can still bypass code-randomization. Specifically, we show that the attacker can infer the location of return-oriented programming gadgets by disclose a large number of code pointers from data memory and combining them with offline knowledge about the applied code randomization. Our attacks highlight the need for hardening code-randomization schemes against memory-disclosure attacks.

In the second part of this chapter we turn our attention to coarse- and fine-grained control-flow integrity schemes. We demonstrate how attackers can chain virtual function calls of C++ applications to bypass coarse-grained control-flow integrity defenses without violating the enforced policies. One might assume that fine-grained control-flow integrity would solve this issue and provide the perfect protection against code-reuse attacks. However, we highlight the importance of binary security analysis by finding how the optimization pass of two open source compilers introduce a security vulnerability into a conceptually secure compiler-based control-flow integrity implementation. Specifically, the optimization pass forced a value, which used during a control-flow integrity check and which is supposed to be only readable, to be temporarily spilled to memory. This gives the attacker a small time window in which she can tamper with this value to bypass fine-grained control-flow integrity. Finally, we bypass control-flow integrity by manipulating the intermediate representation of a just-in-time compiler for JavaScript. As a consequence, the just-in-time compiler generates attacker controlled, hence, bypassing control-flow integrity as well as other just-in-time compiler defenses.

To conclude, both randomization-based and control-flow integrity-based defenses may offer good protection against the vast majority of memory-corruption and code-reuse attacks. However, all it takes for the attacker to succeed is to find one weakness in the defense, and neither randomization- nor integrity-based defenses offer complete protection.

In this chapter, we focus on using code-randomization techniques to mitigate the effectiveness of code-reuse attacks. Code randomization is, besides control-flow integrity, an efficient and effective mitigation against code-reuse attacks. Unfortunately, as we demonstrated in the previous chapter, code-randomization can be bypassed by means of memory-disclosure attacks. Following, we discuss techniques that mitigate the effects of disclosure attacks. Specifically, we demonstrate how execute-only memory can serve as a primitive to mitigate direct and indirect memory-disclosure attacks. In Section 4.1 we leverage memory virtualization to implement execute-only memory. However, not all platforms support memory virtualization. Hence, we explore software-based techniques to implement execute-only memory in Section 4.2. In Section 4.3 we present the design of a linker extension that embeds necessary meta-data and code to produce self-randomizing binaries. Next, we turn our attention toward a method for leveraging randomization to efficiently mitigate data-only attacks against the page table. Specifically, we relocate the page tables to a memory region that provides enough entropy. We also apply leakage resilience to ensure that an attacker, who can leak kernel memory, cannot find references that point to the new memory location. Our results show that randomization-based defenses can be hardened to provide resilience against memory-disclosure attacks. Lastly, we summarize related work on code-reuse defenses in Section 4.5 and conclude this chapter in Section 4.6.

4.1 READACTOR: MEMORY-DISCLOSURE RESILIENT CODE RANDOMIZATION

Today code-reuse attacks are the most prevalent technique for attackers to gain full control of a system. Therefore, attackers exploit a memory-corruption vulnerability to overwrite a code pointer which is then used by the vulnerable application as a target address of an indirect branch. The overwritten code pointer points to a gadget, which is existing application code that performs an attacker-desired task. Despite a substantial amount of research over the last decade the problem of code-reuse attacks remains unsolved. Existing mitigations can roughly be classified as Control-flow Integrity (CFI) and code-randomization-based solutions.

CFI mitigates code-reuse attacks by verifying each code pointer against a policy before it is used as a branch target, hence, limiting the gadgets available to the attacker. The enforced policy is derived from a statically computed control-flow graph (CFG) [6]. The effectiveness of CFI greatly depends on the precision of the CFG, which is hard to compute even when the source code is available [69]. Further, CFI based on a precise CFG often negatively impacts the performance of the protected program. Therefore, recent work on CFI investigated the feasibility of trading the precision of the enforced CFG for increased performance [43, 73, 163, 237, 239]. However, this idea was quickly

rejected because it allows the attacker to construct code-reuse attacks that do not violate the enforced policy, hence, fully bypass the mitigation [31, 60, 83, 84, 185].

Recently, Kuznetsov et al. [118] published Code Pointer Integrity (CPI), which follows a similar approach to CFI. The difference is CPI verifies write accesses to code pointers whereas CFI verifies the usage of code pointers. Therefore, CPI isolates all code pointers from non-control data, as originally suggested by Szekeres et al. [205], by moving it to a special memory area, called safe region. The main challenge for CPI is to protect the safe region from attackers with arbitrary read-write capabilities. Due to the lack of in-process memory isolation schemes on 64-bit architectures, Kuznetsov et al. [118] try to hide the safe region using randomization. Unfortunately, this quickly turned out to be an insufficient protection when Evans et al. [68] bypassed CPI by exploiting a side channel to reveal the location of the safe region.

Code randomization mitigates code-reuse attacks by reorganizing the layout of the application in memory [119] during run time. As a consequence, the addresses of gadgets, which the attacker identified during an offline analysis, become invalid. However, once major applications adopted code randomization, attackers started to utilize memory-disclosure vulnerabilities to leak the randomization secret [189, 203]. As discussed in Section 3.1, we distinguish between *direct* and *indirect* memory-disclosure. The main difference between both types is that in a direct memory-disclosure attack the leakage is based on code pointers that are encoded into instructions residing on code pages. In an indirect memory-disclosure attack the attacker leaks multiple code pointers that reside in data memory, and combines it with offline knowledge about the application and the applied randomization to infer the layout during run time.

Since randomization-based defenses are more efficient and easier to adopt [59, 95, 97, 150, 218], recent work in the area of code randomization focused on increasing its resilience against disclosure attacks. For example, Oxymoron [16] obfuscates code pointers, which are encoded in instructions, whereas Execute-no-Read (XnR) [17] marks code pages, which are currently not being executed, as non-accessible to prevent Just-in-Time Return-oriented Programming (JIT-ROP) [196] attacks. However, as we described in Section 3.1 both techniques cannot mitigate indirect disclosure attacks, and can easily be bypassed.

Goals and contributions. Our main goal is to tackle the challenge of memory disclosure to harden code-randomization based defenses. We use our classification of direct and indirect memory disclosure which we presented in Section 3.1. We present the design and implementation of Readactor, the first practical fine-grained code-randomization defense that resists both classes of memory-disclosure attacks. Readactor utilizes hardware-enforced eXecute-only Memory (XoM) as a trusted computing base. By combining XoM with our novel compiler transformation we achieve resilience against direct and indirect memory disclosure. Contrary to previous work, we do not rely on insecure software emulation [17], or legacy hardware [80] but base our implementation of XoM on virtualization [103] which is supported by commodity Intel CPUs. To summarize, our main contributions are:

- **Comprehensive ROP resilience.** Readactor prevents all existing ROP attacks: conventional ROP [192], ROP without returns [35], and dynamic ROP [22, 196].

Most importantly, Readactor improves the state of the art in JIT-ROP defenses by preventing indirect memory disclosure through code-pointer hiding.

- **Novel techniques.** We introduce compiler transformations that extend execute-only memory to protect against the new class of indirect information disclosure. We also present a new way to implement execute-only memory that leverages hardware-accelerated memory protections.
- **Covering statically & dynamically generated code.** We introduce the first technique that extends coverage of execute-only memory to secure just-in-time (JIT) compiled code.
- **Realistic and extensive evaluation.** We provide a full-fledged prototype implementation of Readactor that diversifies applications, and present the results of a detailed evaluation. We report an average overhead of 6.4% on compute-intensive benchmarks. Moreover, our solution scales beyond benchmarks to programs as complex as Google’s popular Chromium web browser.

We covered the technical background of return-oriented programming (ROP) attacks in Section 2.4, and of memory-disclosure attacks in Section 3.1. In the remainder of this chapter we first introduce our threat model, then provide an overview of the design and implementation of Readactor. We continue with a detailed performance/security evaluation, and conclude with a discussion about potential weaknesses.

4.1.1 Threat Model

Our threat model is consistent with prior offensive and defensive work, particularly the powerful model introduced in JIT-ROP [196].

Defense Capabilities

Writable \oplus Executable. The target system provides built-in protection against code injection attacks. Today, all modern processors and operating systems support data execution prevention (DEP) to prevent code injection.

Secure loading. The attacker cannot tamper with our implementation of Readactor.

Code Randomization. The attacker has no a priori knowledge of the in-memory code layout. We ensure this through the use of fine-grained diversification.

Adversary Capabilities

Known System Configuration. The attacker knows the software configuration and defenses on the target platform, as well as the source code of the target application.

Memory Corruption. The target program suffers from at least from one memory corruption vulnerability which allows the attacker to hijack the control-flow.

Information disclosure. The attacker is able to read and analyze any readable memory location in the target process.

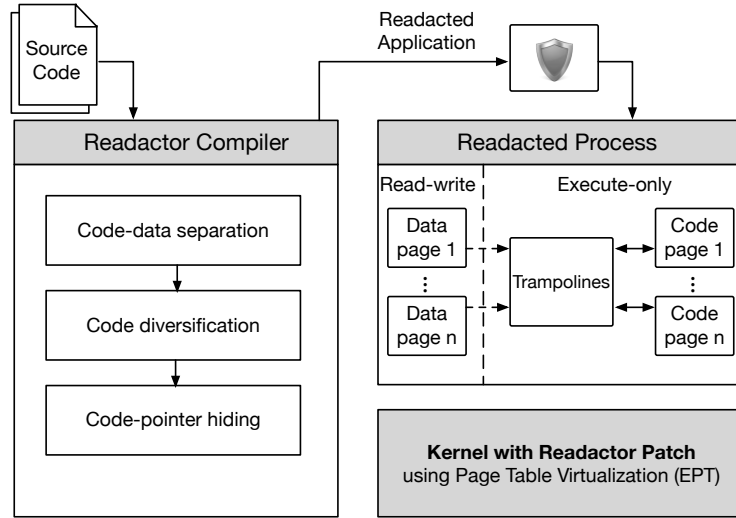


Figure 28: System overview. Our compiler generates diversified code that can be mapped with execute-only permissions and inserts trampolines to hide code pointers. We modify the kernel to use EPT permissions to enable execute-only pages.

We cannot rule out the existence of timing, cache, and fault side channels that can leak information about the code layout to attackers. Although information disclosure through side-channels is outside the scope of this chapter, we note that Readactor mitigates recent remote side-channel attacks against diversified code since they also involve direct memory disclosure [22, 188].

4.1.2 Readactor Design and Implementation

In this section, we first provide an overview of our overall design of Readactor before providing details on its core components: execute-only memory and code-pointer hiding.

4.1.2.1 Overview

Readactor leverages eXecute-only Memory (XoM) as primitive to mitigate direct and indirect memory-disclosure attacks (see Chapter 3.1). We note that x86 does not natively support mapping memory as execute-only. Previous related work [17] tried to overcome this issue by emulating XoM in software, however, this leaves at least the page, which contains the code that is currently executed, readable (in fact, for performance reasons Backes et al. [17] utilize a sliding window which leaves n pages readable). Readactor, on the other hand, utilizes the hardware virtualization support of commodity x86 Central Processing Units (CPUs) to implement XoM. Hence, we prevent the attacker from directly disclosing memory at *any* time during the program execution. We introduce *code-pointer hiding* to protect all code pointers against indirect memory-disclosure attacks. Code-pointer hiding is based on *trampolines*, which are direct jump instructions that are protected using execute-only memory.

Our approach to protecting code pointers requires precise control-flow information to identify all code pointers. Therefore, we opt for a compiler-based approach because binary analysis approaches are too error prone due to the information loss during which occurs during compilation. As we show in Section 4.1.4, this approach allows us to scale Readactor to complex real-world applications like browsers and JavaScript engines without imposing an impractical performance hit.

Figure 28 illustrates the architecture of Readactor. As a first step, our compiler extension creates a *readacted* application in three steps: first, the compiler ensures that code and data are strictly separated from each other. In particular, we prevent the compiler from embedding data, like jump tables, into the code section, which is normal done for performance reasons, and ensure that code and data sections start on a new page, and are not appended to each other. Second, we randomize the code layout of the application. There are a large number of randomization strategies [119], however, we found that function permutation [113] and callee-save register slot reordering [162] provides an optimal tradeoff between performance and security. In our current implementation, we randomize the application only during compile time. This assumes that the randomized application remains secret from the attacker. However, this is merely an implementation limitation rather than a conceptual limitation. In fact, in Section 4.3 we designed and implemented a linker wrapper to create self-randomizing binaries. Third, we create a trampoline for every code pointer.

To enable execute-only memory on commodity x86 CPUs, we implemented a thin hypervisor that enables and configures memory virtualization (Extended Page Tables (EPT)), and provides an interface to the operating system kernel to mark single pages as execute-only. Finally, we patch the operating system kernel to support loading readacted applications by mapping their code pages as execute-only by interacting with the hypervisor.

Next, we provide detailed information on the implementation of Readactor’s core components: execute-only memory and code-pointer hiding.

4.1.2.2 *Execute-only Memory*

Execute-only memory is not natively supported by modern operating systems. In the following we discuss how we overcome the challenges of enabling execute-only memory for modern operating systems on the x86 architecture. Therefore, we first provide a short technical background on memory protection before we explain how we leverage a thin hypervisor to enable execute-only memory, and patch the operating system to use it.

Extended Page Tables

The x86 architecture uses multiple layers of indirection to manage access to the memory. During the translation from one layer to another the CPU enforces an access control policy that can be configured through software. Traditionally, x86 uses two indirection layers for memory management: segmentation and paging. Segmentation is a legacy feature, which does not enforce any memory protection on 64-bit systems anymore. Therefore, modern operating systems rely solely on paging to enforce memory

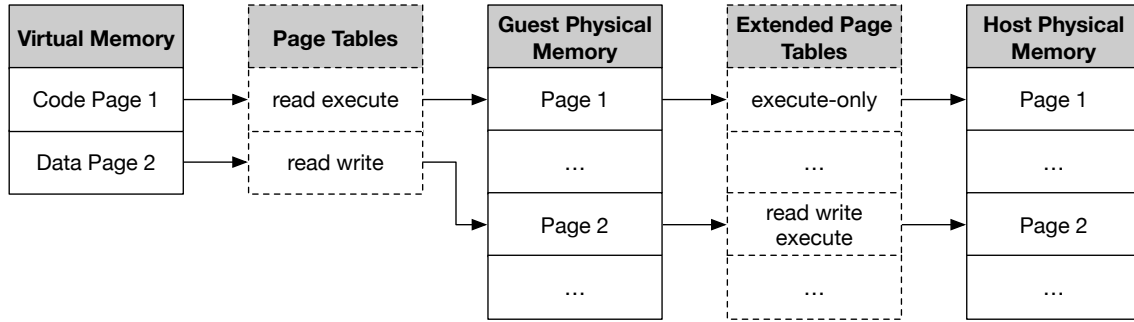


Figure 29: Relation between virtual, guest physical, and host physical memory. Page tables and the EPT contain the access permissions that are enforced during the address translation.

protection. The operating system configures paging through a data structure, called *page table*, which contains the information to translate *virtual addresses* to *physical addresses*, and the corresponding memory permissions. Interestingly, x86 paging was extended to support non-executable memory [10, 103], however, not execute-only memory.

In late 2008 Intel introduced hardware virtualization, which includes memory virtualization. Memory virtualization is implemented by adding another layer of indirection for memory accesses. When virtualization is active, physical addresses, as seen by the operating system, are now called *guest physical addresses*, and are translated to *host physical addresses*, which are the real physical addresses, using the Extended Page Tables (EPT) [103]. The EPT contain, similar to the regular page tables, translation information and memory permissions. However, in contrast to the regular page tables, the EPT do allow the enforcement of (non-)readable, (non-)writable, and (non-)executable memory permissions independently.

Figure 29 illustrates the translation process from virtual memory to physical memory with paging and memory virtualization enabled. Here, the loaded application consists of a code and a data page. The effective permission for both pages is the intersection of the permission sets of the regular *and* the extended page table. For example, if the attacker tries to read from the code page, the first translation using the regular page tables will succeed because, here, the memory protection is set to allow read and execute access. However, the second translation from the guest physical memory to host physical memory will fail, and create a memory-access violation exception because the memory protection is set to only allow execute access.

To enable the EPT we have to enable the hardware virtualization feature of the CPU. Once enabled, the virtualization features (including the EPT) are managed by a piece of software called *hypervisor*. Next, we explain how we created a minimal hypervisor that only enables and configures memory virtualization.

Hypervisor

Readactor requires the implementation or extension of an existing hypervisor [140, 158, 160, 223, 234] to provide an interface to the operating system to mark individual

pages as execute only. For our implementation of Readactor we chose to implement a small stand-alone hypervisor for two reasons: first, the majority of sophisticated attacks require a scripting environment [39, 40, 60, 61, 186, 196], as is typically provided by browsers and document viewers, and target end users that commonly run their software in a non-virtualized environment. Second, implementing a stand-alone hypervisor allows us to measure the run-time overhead of using execute-only memory with greater precision because it avoids the possibility that any overhead is masked by other features of an existing hypervisor.

Although the original purpose of hardware virtualization was to execute multiple operating systems in parallel on a single hardware platform, in mid 2017, Microsoft started to deploy a hypervisor on Windows 10 to enforce security policies as well [142]. This confirms that our approach of a stand-alone hypervisor for end-user systems is indeed practical.

Our implementation of the hypervisor us allows to enable and disable virtualization on-the-fly. This process is fully transparent to the operating systems, and was inspired by previous work [115, 180] which used this approach to load stealth, hypervisor-based rootkits. However, contrary to rootkits, our hypervisor enables only execute-only memory, and provides an interface to the operating system to manage it. The fact that our hypervisor only needs to enable and configure memory virtualization allows us keep the hypervisor as small as 500 lines of code which benefits both security and performance.

The naïve approach to enable execute-only memory would be to create an *identity* EPT mapping where the guest physical address is the same as the host physical address, and to provide a *hypercall*, which is a hypervisor function that can be invoked by the operating system, to change the permission of the EPT mappings. Such an approach comes with two disadvantages: first, it gives the operating system full control of the EPT mapping, hence, the hypervisor is not properly isolated from the operating system. Although the operating system is trusted in our threat model, a solution that allows the hypervisor to isolate itself from the operating system is a desirable design goal. Second, every hypercall requires a context switch from the operating system to the hypervisor and back which negatively impacts the performance of the overall system. We found that by providing two mappings, one *normal* and one *readacted* mapping, the host physical memory avoids both weaknesses of the naïve approach.

Figure 30 shows an example of the EPT mapping: On the far right is the actual physical memory of the system which has a total size of 4 GB in this example configuration. Our hypervisor configures the EPT such that from the operating system’s point of view, the system appears to have 20 GB available. However, the address 0 GB and 16 GB of the guest physical memory both map to the same physical memory page. The difference is, that for the address 0 GB the permissions in the EPT are set to *read-write-execute* whereas for the address 16 GB the permissions are set to *execute-only*. Hence, the effective permissions for the normal mapping are the permissions of the regular page table, and execute-only for the readacted mapping. Note that our hypervisor maps the readacted mapping at an offset that is to the power of two. Hence, the operating system can change

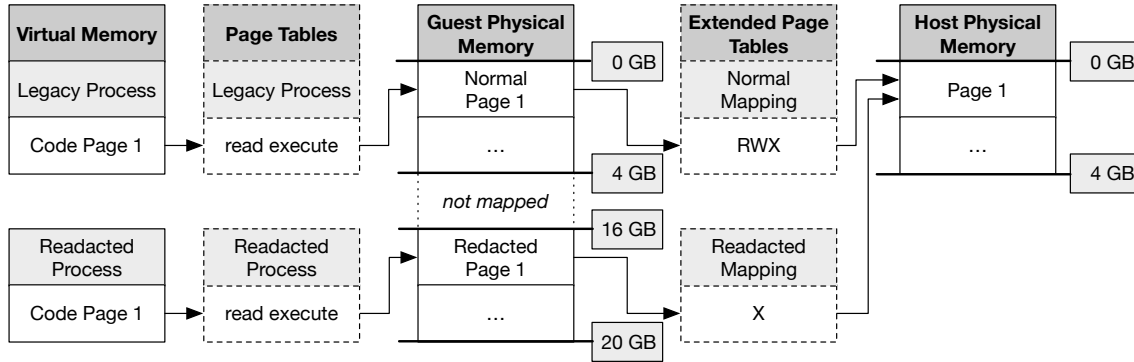


Figure 30: Readactor creates two mappings for each physical memory page: a readacted mapping, which maps the physical memory as execute-only, and a normal mapping which maps the physical memory as read-write-execute. The operating system can map individual pages as execute-only by mapping virtual memory of a process either to the normal or readacted guest physical memory page.

the permission of a virtual memory page by flipping a bit of the corresponding entry in the regular page table.

Further, our design is fully compatible with legacy applications because the normal mapping is used by default, even when Readactor is active. In fact, Figure 30 demonstrates how shared memory between a legacy and a readacted application. Naturally, the legacy does not prevent the attacker from reading the memory, thus disclosing the content of the code page. As a consequence, the attacker could use the gained knowledge to compromise the readacted application. Therefore, while sharing code pages between readacted and legacy applications is possible it is not advised.

Operating System

We extended the Linux kernel to use the interface of our hypervisor. However, the concept of Readactor is operating system agnostic and can be applied to other operating system, like Windows or macOS, as well.

Our patch to the Linux kernel adds an additional 82 lines of code to the Linux kernel. Specifically, it extends the part of the kernel that is responsible for loading binaries: every binary includes a header that defines the permissions for each section of the binary. While the compiler can already set the read-write-execute permission independently for a section in the binary header, the operating system could not map a section as execute-only as the execute and read permission are the same in the x86 page table. However, with our hypervisor in place we patched the Linux kernel to map sections of a binary as execute-only if the permission is set in the binary header.

4.1.2.3 Code-pointer Hiding

Ideally, two instances of a fine-grained code-randomization hardened application do not share *any* common gadgets. In such cases, execute-only memory offers sufficient protection against memory-disclosure attacks because the attacker can neither read,

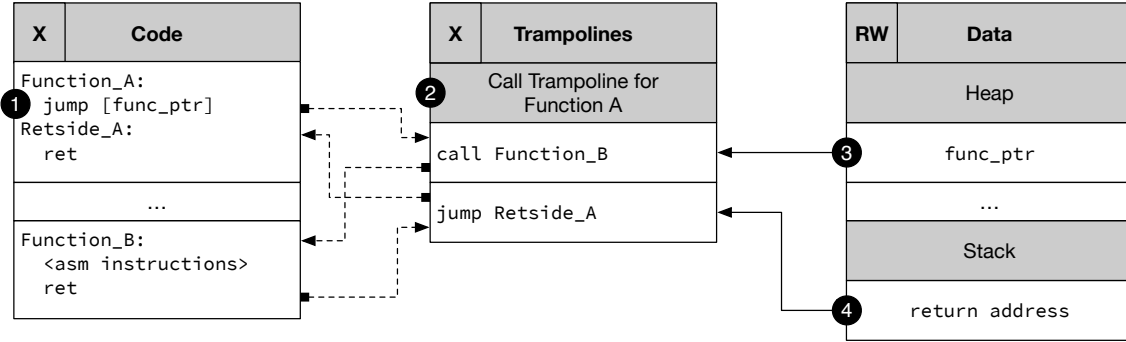


Figure 31: Readacted applications replace code pointers in readable memory with trampoline pointers. The trampoline layout is not correlated with the function layout. Therefore, trampoline addresses do not leak information about the code to which they point.

nor infer the memory content through an indirect disclosure attack. However, ideal fine-grained code randomization is not practical due to increased run-time and memory overhead [119]. Therefore, practical fine-grained randomization schemes rely on coarser granularity. In an indirect memory-disclosure attack (cf. Section 3.1), the attacker exploits that two instances of code-randomization-hardened applications share common gadgets. In particular, the attacker performs an offline analysis to find gadgets where the offset to a code pointer remains constant after the application is randomized.

With code-pointer hiding we present a technique that relaxes the requirement of ideal fine-grained code randomization by ensuring that the offsets between code pointers and gadgets are no longer constant. We achieve this by creating a layer of indirection for code pointers which is protected through execute-only memory. Specifically, we create so-called *trampoline* for each code pointer. A trampoline is a direct jump instruction, which encodes the value of a code pointer in the jump instruction. All trampolines are located in execute-only memory, thus, protected from disclosing the jump target through direct disclosure attack.

Figure 31 illustrates a call trampoline. First, the call instruction in Function_A is substituted with a jump instruction because a call instruction writes the return address, which is the address of the next instruction after the call, on the stack that would allow the attacker to disclose the location of the code section ❶. Next, our compiler extension generates a trampoline for each call site that consists of a direct call to the original target, and a jump to the return side ❷. As a consequence, the return instruction, which is pushed onto the stack during the call of Function_A, points to the trampoline section, hence, revealing no information about the code section ❸. Finally, our compiler extension ensures that function pointers are substituted with pointers to the corresponding trampoline ❹.

4.1.2.4 Code-pointer Hiding for Function Tables

Our design for trampolines prevents the attacker from performing indirect-disclosure attacks. However, for Counterfeit Object-oriented Programming (COOP) attacks (cf. Section 3.3) the attacker reuses whole functions instead of small instruction sequences.

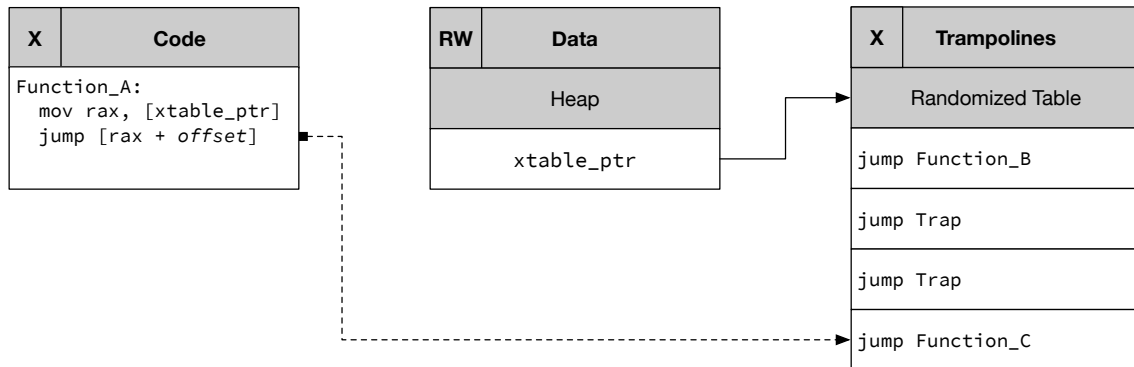


Figure 32: In readacted applications, the function pointer tables are substituted with trampolines. Further, their entries are randomized, and, to counter brute-force attack on the entropy of the table layout, we insert trampolines to trap functions.

For an unprotected application, the attacker first discloses function pointers of function tables, like virtual tables (vtables) or the Processor Linkage Table (plt). Since in a readacted application each function pointer is substituted with a pointer to a trampoline, which jumps to the corresponding function, the attack can simply reuse trampolines pointers to achieve the same results.

To counter COOP style attacks, we extend Readactor, to protect function tables. Our extension, called Readactor++, converts function tables into trampolines and randomizes the order of the entries, as shown in Figure 32. As in Readactor, call sites are instrumented to prevent disclosing a valid code address through a return address. However, in some cases randomizing the order is not sufficient because the original function-pointer table contains only a few entries. Thus, only randomizing the order leaves the possibility of brute-force attacks. We mitigate brute-force attacks on the table layout by inserting additional entries into the randomized table that redirect the control flow to *trap* functions. Depending on the implementation, trap functions can terminate the whole process tree to prevent brute-force attacks on the table layout [76], or notify an intrusion detection system about an on-going attack.

4.1.3 Security Evaluation

The main goal of Readactor is to prevent code-reuse attacks constructed using either direct or indirect disclosure vulnerabilities. Thus, we have analyzed and tested its effectiveness based on five different variants of code-reuse attacks, namely (1) static ROP attacks using direct and indirect disclosure, (2) Just-in-Time Return-oriented Programming attacks using direct disclosure, (3) Just-in-Time Return-oriented Programming attacks using indirect disclosure, (4) whole-function reuse attacks, such as return-into-libc or COOP. We present a detailed discussion on each type of code-reuse attack and then evaluate the effectiveness of Readactor using a sophisticated proof-of-concept JIT-ROP exploit.

4.1.3.1 *Static ROP*

To launch a traditional ROP attack [35, 192], the attacker must know the run-time memory layout of an application and identify ROP gadgets based on an offline analysis phase. To defeat regular Address Space Layout Randomization (ASLR), the attacker needs to leak a single run-time address through either direct or indirect disclosure. Afterwards, the addresses of all target gadgets can be reliably determined.

Since Readactor performs fine-grained randomization using function permutation, the static attacker can only guess the addresses of the target gadgets. In other words, the underlying fine-grained randomization ensures that the attacker can no longer statically determine the addresses of all gadgets as offsets from the run-time address of a single leaked function pointer. In addition, we randomize register allocation and the ordering of stack locations where registers are saved to ensure that the attacker cannot predict the run-time effects of gadgets. Using these fine-grained diversifications, Readactor fully prevents static ROP attacks.

4.1.3.2 *JIT-ROP with direct disclosure*

JIT-ROP attacks bypass fine-grained code randomization schemes by disassembling code pages and identifying ROP gadgets dynamically at run time. One way to identify a set of useful gadgets for a ROP attack is to exploit direct references in call and jump instructions [196]. Readactor prevents this attack by marking all code pages as non-readable, i.e., execute-only. This differs from a recent proposal, XnR [17], that always leaves a window of one or more pages readable to the attacker. Readactor prevents all reading and disassembly of code pages by design.

4.1.3.3 *JIT-ROP with indirect disclosure*

Preventing JIT-ROP attacks that rely on direct disclosure is insufficient, since advanced attacks can exploit indirect disclosure, i.e., harvesting code pointers from the program's heap and stack (see Section 3.1). Readactor defends against these attacks with a combination of fine-grained code randomization and code-pointer hiding. As stated above, pointer hiding ensures that the attacker can access only trampoline addresses but cannot disclose actual run-time addresses of functions and call sites (see Section 4.1.2.3). Hence, even if trampoline addresses are leaked and known to the attacker, it is not possible to use arbitrary gadgets inside a function because the original function addresses are hidden in execute-only trampoline pages. Code-pointer hiding effectively provides at least the same protection as coarse-grained CFI, since only valid address-taken function entries and call-sites can be reused by an attacker. However, our scheme is strictly more secure, since the attacker must disclose the address of each trampoline from the stack or heap before he can reuse the function or call-site. In addition, we strengthen our protection by employing fine-grained diversifications to randomize the dataflow of this limited set of control-flow targets.

Specifically, when exploiting an indirect call (i.e., using knowledge of a trampoline address corresponding to a function pointer), the attacker can only redirect execution to the trampoline but not to other gadgets located inside the corresponding function.

In other words, we restrict the attacker who has disclosed a function pointer to whole-function reuse.

On the other hand, disclosing a call trampoline allows the attacker to redirect execution to a valid call site (e.g., call-preceded instruction). However, this still does not allow the attacker to mount the same ROP attacks that have been recently been launched against coarse-grained CFI schemes [31, 60, 83, 185], because the attacker only knows the trampoline address and not the actual run-time address of the call site. Hence, leaking one return address does not help to determine the run-time addresses of other useful call sites inside the address space of the application. Furthermore, the attacker is restricted to only those return trampoline addresses that are leaked from the program's stack. Not every return trampoline address will be present on the stack, only those that are actually used and executed by the program are potentially available. This reduces the number of valid call sites that the attacker can target, in contrast to the recent CFI attacks, where the attacker can redirect execution to *every* call site in the address space of the application without needing any disclosure.

Finally, to further protect call-site gadgets from reuse through call trampolines, we use two fine-grained diversifications proposed by Pappas et al. [162] to randomize the dataflow between gadgets: register allocation and stack slot randomization. Randomizing register allocation causes gadgets to have varying sets of input and output registers, thus disrupting how data can flow between gadgets. We also randomly reorder the stack slots used to preserve registers across calls. The program's Application Binary Interface (ABI) specifies a set of callee-saved registers that functions must save and restore before returning to their caller. In the function epilogue, the program restores register values from the stack into the appropriate registers. By randomizing the storage order of these registers, we randomize the dataflow of attacker-controlled values from the stack into registers in function epilogues.

4.1.3.4 Whole-Function Reuse Attacks

Our goal is to prevent attacks utilizing whole-function reuse such as return-into-libc and COOP which rely on disclosing function addresses from function pointer tables. With Readactor in place, the attacker can still disclose the addresses of trampolines stored in readable structures. However, as Readactor randomizes the layouts of function tables, identifying the matching trampoline with a function becomes a challenge for the attacker. This leaves the attacker with only the option of guessing the entries in randomized function tables.

As mentioned at the end of Section 4.1.2.3 we can mitigate brute-force attacks by terminating the application after a trap function is hit. Booby traps will not be hit during correct program execution.

Since hitting a booby trap will terminate the attack, a successful attacker needs to make an uninterrupted sequence of correct guesses of entries in the randomized function table. What exactly constitutes a correct guess depends on the concrete attack scenario. In the best case, the attacker must always guess a particular entry in a particular function table; in the worst case, a good guess for the attacker may be any entry that is not a booby trap. Considering the nature of existing COOP and

return-to-libc attacks [186, 216], we believe that the former case is the most realistic. Further, assuming in favor of the attacker that she will only attempt to guess entries in tables with exactly 16 entries (the minimum), we can roughly approximate the probability for Readactor to prevent an attack that reuses n functions with $P \approx 1 - (\frac{1}{16})^n$. Our experiments in the following indicate that an attacker needs at least two or three hand-picked functions (most-likely from distinct tables) to mount a successful return-to-libc (RILC) or COOP attack respectively. Thus, the probability of preventing these attacks is lower bounded by $P_{\text{RILC},\min} \approx 1 - (\frac{1}{16})^2 = 0.9960$ and $P_{\text{COOP},\min} \approx 1 - (\frac{1}{16})^3 = 0.9997$.

4.1.3.5 *Proof-of-concept exploit*

To demonstrate the effectiveness of our protection, we introduce an artificial vulnerability into V8 that allows an attacker to read and write arbitrary memory. This vulnerability is similar to a vulnerability in V8¹ that was used during the 2014 Pwnium contest to get arbitrary code execution in the Chrome browser. In an unprotected version of V8, the exploitation of the introduced vulnerability is straightforward. From JavaScript code, we first disclose the address of a function that resides in the JIT-compiled code memory. Next, we use our capability to write arbitrary memory to overwrite the function with our shellcode. This is possible because the JIT-compiled code memory is mapped as RWX in the unprotected version of V8. Finally, we call the overwritten function, which executes our shellcode instead of the original function. This attack fails under Readactor, because the attacker can no longer write shellcode to the JIT-compiled code memory, since we set all JIT-compiled code pages as execute-only. Further, we prevent any JIT-ROP like attack that first discloses the content of JIT-compiled code memory, because that memory is not readable. We test this by using a modified version of the attack that reads and discloses the contents of a code object. Readactor successfully prevents this disclosure by terminating execution of the JavaScript program when it attempted to read the code.

4.1.4 *Performance Evaluation*

We rigorously evaluated the performance impact of Readactor on both the SPEC CPU2006 benchmark suite and a large real-world application, the Chromium browser. Finally, we measure the performance of Readactor++ independently.

4.1.4.1 *SPEC CPU2006*

The SPEC CPU2006 benchmark suite contains CPU-intensive programs, which are ideal for testing the worst-case overhead of our compiler transformations and hypervisor. To fully understand the impact of each of the components that make up the Readactor system, we measure and report their performance impact independently.

¹ CVE-2014-1705

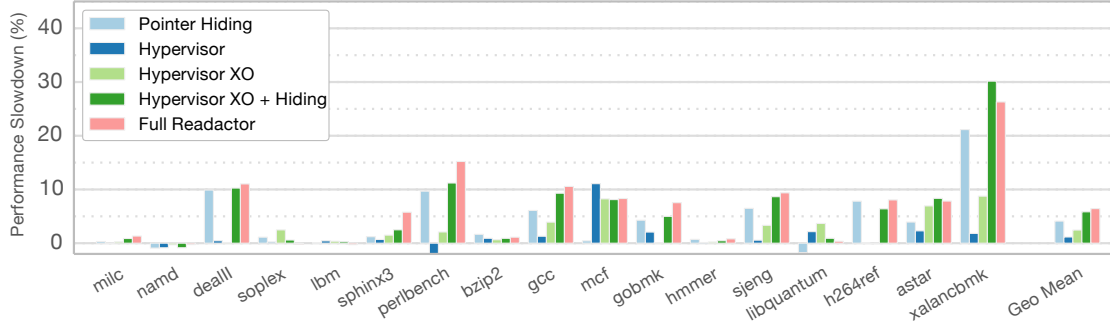


Figure 33: Performance overhead for SPEC CPU2006 with Readactor enabled relative to an unprotected baseline build.

We perform all evaluations using Ubuntu 14.04 with Linux kernel version 3.13.0. We primarily evaluate SPEC on an Intel Core i5-2400 desktop CPU running at 3.1 GHz with dynamic voltage and frequency scaling (Turbo Boost) enabled. We also independently verify this evaluation using an Intel Xeon E5-2660 server CPU running at 2.20 GHz with Turbo Boost disabled, and observe identical trends and nearly identical performance (within one percent on all averages). We summarize our SPEC measurements in Figure 33.

Enabling code-pointer hiding along with page protections provided by the hypervisor results in a slowdown of 5.8% (*Hypervisor XO + Hiding* in Figure 33). This overhead is approximately the sum of the overheads of both components of the system, the execute-only hypervisor enforcement and pointer hiding. This confirms our hypothesis that each component of the Readactor system is orthogonal with respect to performance.

With the addition of our fine-grained diversity scheme (function, register, and callee-saved register slot permutation) we now have all components of Readactor in place. For the final integration benchmark, we build and run SPEC using three different random seeds to capture the effects of different code layouts. Altogether we observe that the full Readactor system incurs a geometric mean performance overhead of 6.4% (*Full Readactor* in Figure 33). This shows the overhead of applying our full protection scheme to a realistic worst-case scenario of CPU-intensive code, which bounds the overhead of our system in practice.

4.1.4.2 Chromium Browser

To test the performance impact of our protections on complex, real-world software, we compile and test the Chromium browser, which is the open-source variant of Google’s Chrome browser. Chromium is a highly complex application, consisting of over 16 million lines of code [23]. We are easily able to apply all our protections to Chromium with the few minor changes described below. Overall, we find that the perceived performance impact on web browsing with the protected Chromium, as measured by Chromium’s internal UI smoothness benchmark, is 4.0%, which is in line with the average slowdown we observe for SPEC.

To understand the perceived performance impact during normal web browsing we benchmark page scrolling smoothness with Chromium’s internal performance testing framework. We run the scrolling smoothness benchmark from the Chromium source tree on the Top 25 sites selected by Google as representatives of popular websites.

Overall, we find that the slowdown in rendering speed for our full Readactor system was about 4.0%, averaged over 3 different diversified builds of Chromium. This overhead is slightly lower than what we found for SPEC, which is natural considering that browser rendering is not as CPU-intensive as the SPEC benchmarks. However, browser smoothness and responsiveness are critical factors for daily web browsing, rather than raw computing performance.

We also evaluated the performance impact of our techniques on Chromium using the extensive Dromaeo benchmark suite to give a worst-case estimate for browser performance.

We found that execute-only code protection alone, without code-pointer hiding, introduces a 2.8% overall performance slowdown on Dromaeo. Combining the hypervisor execute-only code pages along with code-pointer hiding results in a 12% performance slowdown. However, Dromaeo represents a worst-case performance test, and rendering smoothness on real websites is a far more important factor in browsing.

4.1.4.3 *Readactor++ Extension*

We further, evaluate the performance of our extension Readactor++ on computationally-intensive code with virtual function dispatch using the C++ benchmarks in SPEC CPU2006. Overall, we find that Readactor++ introduces a minor overhead of 1.1%. We measure this slowdown independently of the slowdown introduced by the Readactor itself, which depends on the protection system used and whether hardware natively supports execute-only memory. For a complete system evaluation, we also use the Readactor system to enforce execute-only memory and code-pointer hiding. However, even with this additional slowdown, we find that Readactor++ is competitive with alternative mitigations with an average overhead of 8.4% on SPEC, while offering increased security.

4.1.5 *Discussion: Trampoline-based attacks*

We use trampolines to hide code pointers. However, the addresses of individual trampolines are still exposed in readable memory. To mitigate trampoline-reuse attacks, we randomize the instructions at the trampoline destination, i.e., register allocation randomization and callee-saved register save slot reordering. Currently, we randomize only the used registers, hence, trampoline destinations where only a few registers are used are randomized with a low entropy compared to trampoline destinations that use many of registers. The attacker could exploit the low entropy and attack the randomization using a brute-force attack. While we could increase the entropy by adding more registers this would have a negative impact on the run-time performance.

Further, the attacker can reuse the trampolines to launch whole-function reuse attacks. As described in Section 4.1.2.4, we mitigate whole-function reuse attacks by hardening function pointer tables, like vtables and the plt, with trampolines. We find that in certain cases the attacker can exploit single function pointers in data structures to execute whole-function reuse attacks [179]. Our investigation shows that such attacks could be mitigated using pointer authentication schemes [131, 179].

4.1.6 Conclusion

Previous research demonstrated that code randomization is a practical and efficient mitigation against code-reuse attacks. However, memory disclosure poses a threat to all these probabilistic defenses. Without resistance to such leaks, code randomization loses much of its appeal. This motivates our efforts to construct a code randomization defense that is not only practical but also resilient to all recent bypasses.

We built a fully-fledged prototype system, Readactor, to prevent attackers from disclosing the code layout directly by reading code pages and indirectly by harvesting code pointers from the data areas of a program. We prevent direct disclosure by implementing hardware-enforced execute-only memory and prevent indirect disclosure through code-pointer hiding.

Our careful and detailed evaluation verifies the security properties of our approach and shows that it scales beyond simple benchmarks to complex, real-world software such as Google’s Chromium web browser. Compared to prior JIT-ROP mitigations, Readactor provides comprehensive and efficient protection against direct disclosure, is the first defense to address indirect disclosure, and is also the first technique to provide uniform protection for both statically and dynamically compiled code.

4.2 LR²: SOFTWARE-BASED EXECUTE-ONLY MEMORY

The recent “Stagefright” vulnerability exposed an estimated 950 million Android systems to remote exploitation [63]. Similarly, the “One Class to Rule them All” [168] zero-day vulnerability affected 55% of all Android devices. These are just the most recent incidents in a long series of vulnerabilities that enable attackers to mount code-reuse attacks [151, 177] against mobile devices. Moreover, because these devices run scripting capable web browsers, they are also exposed to sophisticated code-reuse attacks that can bypass ASLR and even fine-grained code randomization by exploiting information-leakage vulnerabilities [45, 61, 188, 196]. Just-in-time attacks (JIT-ROP) [196] are particularly challenging because they misuse run-time scripting to analyze the target memory layout after randomization and relocate a return-oriented programming (ROP) payload accordingly.

There are several alternatives to code randomization aimed to defend against code-reuse attacks, including control-flow integrity (CFI) [6] and code-pointer integrity (CPI) [118]. However, these defenses come with their own set of challenges and tend to have high worst-case performance overheads. We focus on code randomization techniques since they are known to be efficient [59, 94] and scalable to complex, real-world applications such as web browsers, language runtimes, and operating system kernels without the need to perform elaborate static program analysis during compilation.

Recent code randomization defenses offer varying degrees of resilience to JIT-ROP attacks [17, 21, 51, 52, 61, 80, 124, 143]. However, all of these approaches target x86 systems and are, for one reason or another, unfit for use on mobile and embedded devices, a segment which is currently dominated by ARM processors. This motivates our search for randomization frameworks that offer the same security properties as the state-of-the-art solutions for x86 systems while removing the limitations, such as dependence on expensive hardware features, that make them unsuitable for mobile and embedded devices.

The capabilities of mobile and embedded processors vary widely. For instance, many micro-processors do not have a full memory management unit (MMU) with virtual memory support. Instead they use a memory protection unit (MPU) which saves space and facilitates real-time operation². Processors without an MMU can therefore not support defenses that require virtual memory support [17, 51, 52, 80]. High-end ARM processors contain MMUs and therefore offer full virtual memory support. However, current ARM processors do not support³ execute-only memory (XoM) [1] which is a fundamental requirement for randomization-based defenses offering comprehensive resilience to memory disclosure [51, 52].

Therefore, our goal is to design a leakage-resilient layout randomization approach, dubbed LR², that enforces XoM purely in *software* making our technique applicable to MMU-less hardware as well. Inspired by software-fault isolation techniques (SFI) [184,

² MPUs can still enforce W \oplus X policies for a given address range.

³ Firmware executed from non-volatile storage can be marked as execute-only. Code executing out of RAM cannot be marked execute-only on current processors.

187, 225], we enforce XoM by masking load addresses to prevent the program from reading from any code addresses. However, software-enforced XoM is fundamentally different from SFI: First, XoM protects trusted code that is executing as intended whereas SFI constrains untrusted code that may use return-oriented programming techniques to execute instruction sequences in an unforeseen manner to break isolation of the security sandbox. We take advantage of these completely different threat models to enforce XoM in software using far fewer load-masking instructions than any SFI implementation would require; Section 4.2.2.2 provides a detailed comparison. A second key difference between SFI approaches and LR² is that we hide code pointers because they can otherwise lead to indirect leakage of the randomized code layout. Code pointers reveal where functions begin and return addresses reveal the location of call-preceded gadgets [60, 83]. We protect pointers to (virtual) functions (forward pointers) by replacing them with pointers to trampolines (direct jumps) stored in XoM [51]. We protect return addresses (backward pointers) using an optimized pointer encryption scheme that hides per-function encryption keys on XoM pages.

Thanks to software-enforced XoM, LR² only requires that the underlying hardware provides code integrity by enforcing a writable XOR executable ($W \oplus X$) policy. This requirement is met by all recent ARM processors whether they have a basic MPU or a full MMU. Support for $W \oplus X$ policies is similarly commonplace in recent MIPS processors.

In summary, our contributions are:

- LR², the first leakage-resilient layout randomization defense that offers the full benefits of execute-only memory (XoM) without any of the limitations making previous solutions bypassable or unsuitable for mobile devices. LR² prevents *direct* disclosure by ensuring that adversaries cannot use load instructions to access code pages and prevents *indirect* disclosure by hiding return addresses and other pointers to code.
- An efficient return address hiding technique that leverages a combination of XoM, code randomization, XOR encryption, and the fact that ARM and MIPS processors store return addresses in a *link register* rather than directly to the stack.
- A fully-fledged prototype implementation of our techniques capable of protecting Linux applications running atop ARM processors.
- A detailed and careful evaluation showing that LR² defeats a real-world JIT-ROP attack against the Chromium web browser. Our SPEC CPU2006 measurements shows an average overhead of 6.6% which matches the 6.4% overhead for a comparable virtualization-based x86 solution [51].

4.2.1 Threat Model

We use the following threat model:

Defense Capabilities

Secure Loading. The attacker cannot compromise the protected program at compile or load-time. Therefore, the attacker has no a priori knowledge of the code layout.

Writable \oplus Executable. The underlying hardware enforces a W \oplus X policy which prevents code injection. Note that even low-end devices that have an MPU (rather than an MMU) are able to meet this requirement.

Hardware Attacks. Attacks against the underlying hardware or operating system fall outside the scope of this chapter. This includes any attack that uses timing, cache, virtual machine, or fault side channels to disclose the code layout.

Adversary Capabilities

Memory Corruption. At run time, the attacker can read and write data memory such as the stack, heap and global variables. This models the presence of memory corruption errors that allow control-flow hijacking and information leakage.

Our threat model is consistent with prior research on leakage-resilient layout randomization [17, 21, 51, 52, 61, 80].

4.2.2 LR²

Like similar defenses, LR² consists of a series of code transformations. We prototype these transformations as compiler passes operating on source code. Compile-time transformation is not fundamental to our solution. The same approach could be applied by rewriting the program on disk or as it is being loaded into memory.

We perform the following transformations:

- **Load masking** to enforce XoM in software (Section 4.2.2.1). XoM prevents direct disclosure of the code layout and forms the basis for the following transformations. We describe conventional and novel optimizations for efficient instrumentation in Section 4.2.2.2.
- **Forward-pointer hiding** (Section 4.2.2.3). We replace forward pointers to (virtual) functions with pointers into an array of trampolines, i.e., direct jumps to the original pointer address, stored in XoM to prevent *indirect* disclosure similar to Crane et al. [51].
- **Return-address hiding** (Section 4.2.2.4). While we could have hidden return addresses in the same way as we hide forward code pointers, this approach is sub-optimal. First, the return address trampolines (a call and a jump) take up more space than trampolines for forward code pointers (a single jump). Second, this naive approach would require a trampoline between each caller and callee which further increases the memory overhead.
- **Fine-grained code randomization** (Section 4.2.2.5). The preceding techniques prevent disclosure of the code layout, so we must evaluate our system in conjunction with fine-grained diversity techniques.

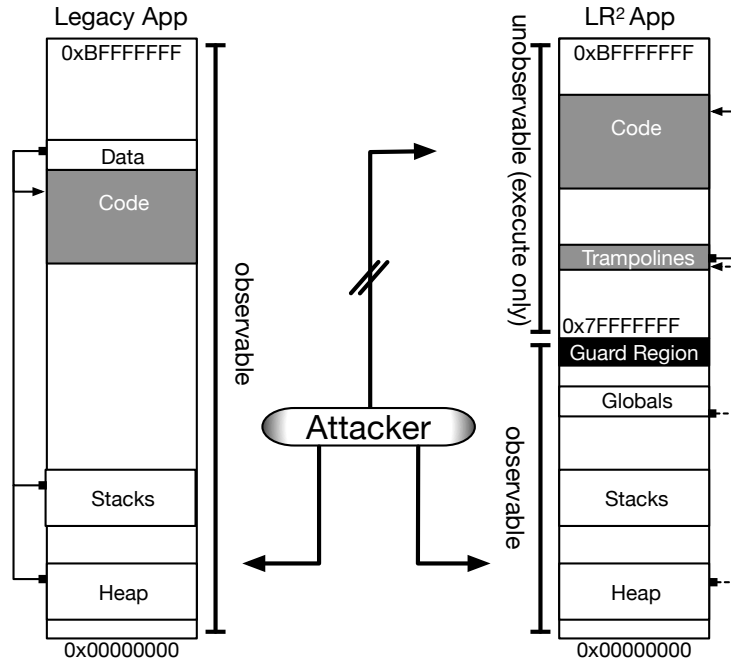


Figure 34: Left: In legacy applications, all pages are observable by attackers. The stack, heap and global areas contain pointers that disclose the location of code pages. Right: In LR² applications, attackers are prevented from observing the upper half of the address space which contains all code. Moreover, attacker observable memory only contains trampoline pointers (dotted arrows) that do not disclose code locations. Finally, return addresses on the stack are encrypted (not shown).

We describe each of these components in detail in the following subsections, along with our prototype LLVM-based toolchain, including dynamic loading and full protection of system libraries.

4.2.2.1 Software-Enforced XoM

On ARM and other RISC instruction sets, all reads from memory use a load instruction (`ldr` on ARM). To enforce XoM purely in software (to avoid reliance on MMU features), we prevent all memory loads from reading program code. We enforce this property by 1) splitting the program code and data memory into separate memory regions, and 2) by ensuring that no load instruction can ever access the code region. We mask every attacker-controlled address that may be used by a load instruction to prevent it from addressing a code page.

We split the virtual memory address space into two halves to simplify load address masking; data resides in the lower half of the address space and code in the upper half (see the right side of Figure 34). Note that we include a *guard region* which consists of 2 memory pages marked as non-accessible. The guard region allows us to optimize loads that add a small constant offset to a base address. With this split, our run-time instrumentation simply checks the most significant bit (MSB) of the address to determine

whether it points to data or code. All valid data addresses (and thus all safe memory loads) must have a zero MSB.

Since we enforce a memory-access policy rather than program integrity in the face of memory corruption, we can optimize our checks to fail safely if the program attempts to read a code address. The ARM instruction set has two options we can use to enforce efficient address checks: the bit clear instruction (`bic`) or a test instruction (`tst`) followed by a predicated load. Either clearing or checking the MSB of the address before a load ensures that the load will never read from the code section. The program may still behave incorrectly if the attacker overwrites an address, but the attacker cannot read any execute-only memory.

The following code uses `bic` masking instrumentation which clears the MSB of the address before accessing memory. This instrumentation is applicable to all load instructions.

```
bic    r0, r0, #0x80000000
ldr    r1, [r0]
```

Listing 9: `bic` masking example

The `tst` masking shown below instead avoids a data dependency between the masking instruction and the load by predicating the load based on a test of the MSB of the address. If an attacker has corrupted the address to point into the code section, the load will not be executed at all since the test will fail. The `tst` masking has the added benefit that we can handle failure gracefully by inserting instrumentation which jumps to an address violation handler in case of failure. However, `tst` is not applicable to loads which are already predicated on an existing condition. In addition, we found that the `bic` masking is up to twice as efficient as `tst` masking on our test hardware, even with the data dependency. One possible reason for this is that the predicated instruction will be speculatively executed according to the branch predictor, causing a pipeline discard in the case of a misprediction. At the same time, `bic` masking benefits greatly from out-of-order execution if the load result is not immediately required.

```
tst    r0, #0x80000000
ldreq  r1, [r0]
```

Listing 10: `tst` masking example

4.2.2.2 *Optimized Load Masking*

Masking addresses before every load instruction is both redundant and inefficient as many loads are provably safe. To optimize our instrumentation, we omit checks for loads that we can guarantee will never read an unconstrained code address. We start with similar optimizations to previous work, including optimizations adapted specifically for ARM, and then discuss a novel optimization opportunity that is not applicable to any SFI technique.

SFI-Inspired Optimizations

We perform several optimizations mentioned by Wahbe et al. [225] in their seminal work on SFI. We allow base register plus small constant addressing by masking only the base register, avoiding the need for an additional address computation add instruction. We also allow constant offset stack accesses without needing checks by ensuring that the stack pointer always points to a valid address in the data section. All stack pointer modifications with a non-constant operand are checked to enforce this property.

Additionally, we do not constrain program counter relative loads with constant offsets. ARM does not allow for 32-bit immediate instructions operands, and therefore large constants are stored in a constant pool allocated after each function. These constant pools are necessarily readable data in the code section, but access to the constant pool is highly constrained. All constant pool loads use a constant offset from the current program counter and therefore cannot be used by attackers to access the surrounding code.

XoM-Specific Optimizations

Although software XoM is inspired by SFI, the two techniques solve fundamentally different problems. SFI isolates potentially malicious code whereas software XoM constrains benign code operating on potentially malicious inputs. In other words, SFI must operate on the assumption that the attacker is already executing untrusted code in arbitrary order whereas software XoM trusts the code it instruments and therefore assumes that the control-flow has not yet been hijacked.

Since we trust the executing code, we can make optimizations to our software XoM implementation that are not applicable when performing traditional SFI load masking. Specifically, we do not need to mask load addresses directly before the load instruction. Instead, we insert the masking operation directly after the instructions that compute the load address. In many cases, a single masking operation suffices to protect multiple loads from the same base address. Registers holding the masked address may be spilled to the stack by the register allocator. Since the stack contents are assumed to be under attacker control (Conti et al. [45] recently demonstrated such an attack), we re-mask any addresses that are loaded from the stack. In contrast, SFI requires that address checks remain in the same instruction bundle as their use, so that a malicious program may not jump between the check and its use. In our experiments, the ability to hoist masking operations allows us to insert 43% fewer masking operations relative to SFI policies that must mask each potentially unsafe load in untrusted code. Figure 35 shows an example in which we are able to remove a masking operation in a loop which substantially reduces the number of bic instructions executed from $2n + 1$ to $n + 1$ where n is the number of loop iterations.

4.2.2.3 Forward-Pointer Hiding

As explained in Section 3.1, adversaries can scan the stack, heap, and static data areas for code pointers that indirectly disclose the code layout. We therefore seek ways to identify functions and return sites without revealing their location. The first major category of

```

1  ; calculate address
2  add ro, ro, r8
3  ; store address on stack
4  str ro, [sp+#12]
5
6  loop:
7  bic ro, ro, #0x80000000
8  ; load address
9  ldr r1, [ro]
10 bic ro, ro, #0x80000000
11 ; load + constant offset
12 ldr r2, [ro+#4]
13 add ro, ro, #8
14 ; check loop condition
15 cmp ro, r3
16 bne loop
17
18 loopend:
19 ; restore address from
20 ; stack, now unsafe
21 ldr ro, [sp+#12]
22 bic ro, ro, #0x80000000
23 ; load address
24 ldr r2, [ro]

```

Software-Fault Isolation

```

1  ; calculate address
2  add ro, ro, r8
3  ; store address on stack
4  str ro, [sp+#12]
5
6  loop:
7  bic ro, ro, #0x80000000
8  ; load address
9  ldr r1, [ro]
10
11 ; load + constant offset
12 ldr r2, [ro+#4]
13 add ro, ro, #8
14 ; check loop condition
15 cmp ro, r3
16 bne loop
17
18 loopend:
19 ; restore address from
20 ; stack, now unsafe
21 ldr ro, [sp+#12]
22 bic ro, ro, #0x80000000
23 ; load address
24 ldr r2, [ro]

```

Software XoM

Figure 35: Differences between load-masking for software-fault isolation (left) and software-enforcement of XoM (right). Because SFI must consider existing code malicious, it must mask load addresses directly before every use. In contrast, software XoM is protecting trusted code executing legitimate control-flow paths, and can therefore use a single masking operation to protect multiple uses.

code pointers are function pointers, used by the program for indirect function calls. Closely related are basic block addresses used in situations such as switch case tables. We handle all forward code pointers in the same manner but use a special, optimized scheme for return addresses as explained in the following section.

We protect against an attacker using forward code pointers to disclose code layout by indirecting all code pointers through a randomized trampoline table, as proposed by Crane et al. [51]. For each code location referenced by a readable code pointer, we create a trampoline consisting of a direct jump to the target address. We then rewrite all references to the original address to refer instead to the trampoline. Thus, the trampoline address, rather than the function address, is stored in readable memory. We randomize trampoline ordering to remove any correlation between the address of the trampoline (potentially available to the attacker) and the actual code address of the target. Hence, even if an attacker leaks the address of a trampoline, it does not reveal anything about the code layout.

4.2.2.4 *Return-Address Hiding*

In principle, we could hide return addresses using the same trampoline mechanism that we use to protect forward pointers. However, the return address trampolines used by Crane et al. [51] require two instructions rather than the single direct jump we use for forward pointers. At every call site, the caller jumps to a trampoline containing 1) the original call instruction, and 2) a direct jump back to the caller. This way, the return address that is pushed on the stack points into a trampoline rather than a function. However, due to the direct jump following the call, every call site must use a unique return address trampoline.

Return addresses are extremely common. Thus, the extra trampoline indirections add non-trivial performance overhead. Additionally, code size is critical on mobile devices. For these reasons, we take an alternative approach. Due to the way ARM and other RISC instruction sets perform calls and returns, we can provide significantly stronger protection than the return address trampolines of Crane et al. [51] without expensive trampolines for each call site. We build upon the foundation of XoM to safely secure an unreadable, per-function key to encrypt every return address stored on the stack.

While x86 call instructions push the return address directly onto the stack, the branch and link instruction (bl) on ARM and other RISC processors instead places the return address in a link register. This gives us an opportunity to encrypt the return address when it is spilled onto the stack⁴. We XOR all return addresses (stored in the link register) before they are pushed on the stack similarly to the PointGuard approach by Cowan et al. [49]. PointGuard, however, uses a much weaker threat model. It assumed that the attacker cannot read arbitrary memory. In our stronger attacker model (see Section 4.2.1), we must prevent the attacker from disclosing or deriving the stored XOR keys. We therefore use a per-function key embedded as a constant in the code which, thanks to XoM, is inaccessible to adversaries at run time. In our current implementation, these

⁴ Leaf functions do not need to spill the return address onto the stack.

```

1 function:
2     ldr    r12, .FUNCTION_KEY
3     eor    lr, lr, r12
4     push   {lr}
5
6 [ function contents here ]
7
8     pop    {lr}
9     ldr    r3, .FUNCTION_KEY
10    eor    lr, lr, r3
11    bx     lr
12
13 .FUNCTION_KEY: ; constant pool entry, embedded
14 .long      ; in non-readable memory
15     0xeb6379b3

```

Listing 11: Return-address hiding example. Note that constant pool entries are embedded in non-readable memory, as described in Section 4.2.2.2.

keys are embedded at compile time. As this might be vulnerable to offline analysis, we are currently working on extending LR² to randomize the keys at load time.

Listing 11 shows an example of our return-address hiding technique. Line 2 loads the per-function key for the current function, and on line 3 it is XORed into the current return address before this address is spilled to the stack in line 4. Lines 8-11 replace the normal `pop {pc}` instruction used to pop the saved return address directly into the program counter. On lines 8-10, the encrypted return address is popped off the stack and decrypted, and on line 11 the program branches to the decrypted return address.

Considering the advantages of protecting return addresses using XOR encryption, the question arises whether forward pointers can be protected with the same technique. An important difference between forward pointers and return addresses is that the former may cross module boundaries. For instance, an application protected by LR² may pass a pointer to an unprotected library or the OS kernel to receive callbacks. The trampoline mechanism used for forward pointers ensures transparent interoperability with unprotected code while XOR encryption does not without further instrumentation, since legacy code would not know that forward pointers are encrypted. In practice, function calls and returns occur more frequently than forward pointer dispatches, so optimizing return address protection is far more important.

Exception Handling

Itanium ABI exception handling uses stack unwinding and matches call sites to exception index tables. Since our return-address hiding scheme encrypts call site addresses on the stack, stack unwinding will fail and break exception handling. All indirect disclosure protections which hide return addresses from an attacker will be similarly incompatible with stack unwinding, which depends on correctly mapping return addresses to stack frame layout information.

We modified LLVM's stack unwinding library implementation `libunwind` to handle encrypted return addresses. Since the first return address is stored in the link register, the stack unwinder can determine the first call site. From the call site, the stack unwinder is able to determine the function and read the XOR key that was used to encrypt the next return address using a whitelisted memory load. By recursively applying this approach, the unwinder can decrypt all return addresses until it finds a matching exception handler. This approach requires that we trust that the unwinding library does not contain a memory disclosure bug.

4.2.2.5 *Fine-Grained Code Randomization*

LR² does not depend on any particular type of code randomization and can be combined with most of the diversifying transformations in the literature [119]. We choose to evaluate our approach using a combination of function permutation [113] and register-allocation randomization [51, 162] as both transformations add very little run-time overhead. As Backes and Nürnberger [16] point out, randomizing the layout at the level of code pages may help allow sharing of code pages on resource-constrained devices. Note that had we only permuted the function layout, adversaries may be able to harvest trampoline pointers and use them to construct an attack without knowing the code layout. Because these pointers only target function entries and return sites (instructions following a call) this constrains the available gadgets much like a coarse-grained CFI policy would. Therefore, we must assume that gadget-stitching attacks [60, 83] are possible. However, stitching gadgets together is only possible with precise knowledge of how each gadget uses registers; register randomization therefore helps to mitigate such hypothetical attacks.

4.2.2.6 *Decoupling of Code and Data Sections*

References between segments in the same ELF object usually use constant offsets as these segments are loaded contiguously. To prevent an attacker from inferring the code segment base address in LR², we replace static relocations that are resolved during link time with dynamic relocations. This allows us to load the segments independently from each other, because the offsets are adjusted at load time. By entirely decoupling the code from the data section we prevent the attacker from inferring any code addresses from data addresses. As a convenient side-effect of this approach, code randomization is possible without the need for position-independent code (PIC). PIC is necessary to make applications compatible with ASLR by computing addresses relative to the current program counter (PC). Since we replace all PC-relative offsets with absolute addresses to decouple the code and data addresses, we observed slightly increased performance relative to conventional, ASLR-compatible position-independent executables at the cost of slower program loading.

4.2.2.7 *Implementation in LLVM*

We implemented our proof-of-concept transformations for LR² in the LLVM compiler framework. Our approach is not specific to LLVM, however, and is portable to any

compiler or static rewriting framework. However, access to compile-time analysis and the compiler intermediate representation (IR) made our implementation easier. In particular, the mask hoisting optimization described previously is easier at compile time, but not impossible given correct disassembly and rewriting.

Since blindly masking every load instruction is expected to incur a high performance overhead due to the high frequency of load instructions, we take a number of steps to reduce the number of necessary mask instructions. LLVM annotates memory instructions such as loads and stores with information about the type of value that is loaded. We can use this information to ensure that load masking is not applied to loads from a constant address. Such loads are used to access jump table entries, global offset table (GOT) offsets, and other constants such as those in the constant pool. These loads account for less than 2% of all load operations in SPEC CPU2006, so this optimization has a small impact.

LLVM-based SFI implementations (e.g., Sehr et al. [187]) operate purely on the machine instructions late in the backend, roughly corresponding to rewriting the assembly output of the compiler. This makes the insertion of fault isolation instrumentation easier, but misses opportunities for additional optimization that is specific to our load-masking techniques. In order to hoist the masking of potentially unsafe addresses to their definition and avoid redundant re-masking, we leverage static analysis information about the program available earlier in the compiler pipeline. Specifically, we begin by marking unsafe address values while the program values are still in static single assignment (SSA) form [55]. This allows us to easily find the definition of address values used by load instructions, and mask these values. Since stack spilling takes place after this point in the compilation, we must be careful to remask any source addresses restored from the stack, since the attacker may have modified these values while on the stack. In particular, we add markers to values that we mask while the program representation is still in SSA form. During register allocation, we check if marked values are spilled to memory. In the case of spills, we insert a masking instruction when restoring this value from the untrusted stack.

As in Native Client (NaCl) [187], it is necessary to prevent the compiler from generating load instructions using both a base and offset register (known as register-register addressing), to be sure that masking will properly restrict the resulting addresses. We modify the LLVM instruction lowering pass, where generic LLVM IR is converted to machine-specific IR, to prevent register-register addressed instructions. Instead, we insert a separate add instruction to compute the effective address. We make an exception if the load is known to be safe (e.g., a jump table load).

Finally, we insert return address protection instrumentation, stack pointer checks, and trampolines for forward code pointers during compilation as described in the previous sections.

4.2.2.8 Full LR² Toolchain

Code-Data Separation

By masking all load addresses we effectively partition the memory into a readable and unreadable section. Our fully-fledged prototype system uses a slightly modified Linux kernel and dynamic loader to separate the process memory space into readable and unreadable sections (see Figure 34 for an overview of this separation). The kernel and dynamic loader normally load entire ELF objects contiguously. Data segments are usually loaded consecutively above the corresponding module's code. In LR², however, readable segments are placed exclusively in the lower 2GiB region of the process address space, while unreadable (code) segments must be placed in the higher 2GiB region. Consequently, this requires ELF objects to be split. We applied small patches to the Linux kernel (121 LoC) and `musl` dynamic loader (196 LoC) to load each ELF segment into the proper area.

Furthermore, we modified the usual kernel memory mapping mechanism to comply with our memory layout restrictions. By passing an internal flag to `mmap`, an application can specify which memory region the requested memory must be allocated in. This allows the loader to ensure that a program's data segment is mapped low enough in memory that the corresponding executable segment lies between `0x80000000` and `0xC0000000` which is where reserved kernel memory begins. Finally, our patch ensures that memory areas allocated by the kernel (e.g., stacks and heaps) are in the readable region.

We also needed to slightly modify the linker to prepare an executable for use with LR² memory layout. Specifically, we patched the gold linker to *not* mark executable sections as readable⁵ and to assign these sections to high addresses. This type of patch is needed for all XoM solutions, since current linkers mark executable segments with read-execute, rather than execute-only permissions. Additionally, we added linker support for 32-bit offsets in Procedure Linkage Table (PLT) entries, which comes at the cost of one additional instruction per PLT entry. This is necessary because the PLT (unreadable memory) refers to the Global Offset Table (GOT) (readable memory), and therefore might be too far away for the 28-bit address offset previously used.

Libraries

For memory disclosure resilience, all code in an application needs to be compiled with LR², including all libraries. Since the popular C standard library `glibc` does not compile with LLVM/Clang, we tested our implementation with the lightweight replacement `musl` instead. It includes a dynamic loader, which we patched to support our code layout with the same approach as applied to the kernel. We use LLVM's own `libc++` as the C++ standard library, since the usual GNU `libstdc++` depends on `glibc` and GCC.



Figure 36: LR² overhead on SPEC CPU2006. We use the performance of unprotected position independent binaries as the baseline.

4.2.3 Performance Evaluation

We evaluate the performance of LR² using the CPU-intensive SPEC CPU2006 benchmark suite, which represents a worst-case, CPU-bound performance test. We measure the overall performance as well as the impact of each technique in our mitigation independently to help distinguish the various sources of overhead. In addition, we measured the code size increase of our transformations, since code size is an important factor in mobile deployment. Overall, we found that with all protections enabled, LR² incurs a geometric mean performance overhead of 6.6% and an average code size increase of 5.6%. We summarize the performance results in Figure 36. Note that these measurements include results for the *hmmer* and *soplex* benchmarks, which are known to be very sensitive to alignment issues ($\pm 12\%$ and $\pm 6\%$, respectively) [134].

We want to measure the impact of LR² applied to whole programs (including libraries), so we compile and protect a C and C++ runtime library with our modifications for use with the SPEC suite. Since the de-facto standard libraries on Linux, *glibc* and *libstdc++*, don't compile with LLVM/Clang, we use *musl* and LLVM's own *libc++* instead. We extended the *musl* loader to support our separated code and data layout.

The *perlbench* and *namd* benchmarks required small workarounds since they contain *glibc*/*libstdc++* specific code. *h264ref* on ARM fails for unknown reasons when comparing the computation result, both for the unmodified and the LR² run; since it completes the computation we include the running time nonetheless. Finally, the stack unwinding library used by LLVM's *libc++* fails with *omnetpp*, so we exclude it from all benchmark measurements. We report all measurements as the geometric mean over all other SPEC CPU2006 benchmarks. All measurements are from a Chromebook model CB5-311-T6R7 with an Nvidia Tegra Logan K1 System-on-Chip (SoC), running Ubuntu 14.04 with Chromium OS's Linux 3.10.18 kernel.

⁵ Note that the memory permission *execute* normally implies *readable* due to the lack of hardware support

4.2.3.1 *Forward-Pointer Hiding*

We measured impact of forward-pointer hiding, which introduces an additional direct jump instruction for each indirect call. We found that this transformation resulted in an overhead of less than 0.3% on average over all benchmarks, with a maximum overhead of 3%.

4.2.3.2 *Return-Address Hiding*

Return-address hiding requires one extra load and XOR at the entry of each function that spills the link register. At each function return it replaces the return instruction with one load, one XOR and one branch. We found that this instrumentation added an overhead of less than 1% on average, with a maximum overhead of 3% over the baseline time. Combining forward-pointer hiding and return-address hiding, we measured an average overhead of 1.4%. We show the combined results in Figure 36, labeled *Pointer Hiding*. This overhead compares favorably to Readactor's [51] 4.1% overhead for full code pointer hiding, since our return-address hiding scheme does not require expensive return trampolines for each call site.

For both forward-pointer and return-address hiding, we noticed that a few benchmarks ran slightly faster with the instrumentation than without. We attribute this variance to measurement error and slight code layout differences resulting in different instruction cache behavior.

4.2.4 *Register-Register Addressing Scheme Restrictions*

An important feature of the ARM instruction set is register offset addressing for array or buffer loads. As described in Section 4.2.2, we have to disable this feature in LR², since it interferes with XoM address masking. We measured the overhead that this restriction incurs by itself and found that restricting register addressing schemes incurs 2.3% overhead on average and a 9% worst-case overhead on the gobmk benchmark. Benchmarks like hmmer, bzip2 and sjeng are affected because a large portion of the execution time is spent in one hot loop with accesses to many different arrays with varying indices.

4.2.4.1 *Software XoM*

The last component to analyze individually is our XoM instrumentation—masking unsafe loads. We found that, after applying the optimizations outlined in Section 4.2.2.2, software-enforced XoM results in an overhead of 6.6% on average (labeled *Software XoM* in Figure 36), with a maximum overhead of 16.4% for one benchmark, gobmk. We attribute this primarily to data dependencies introduced between the masking and load instructions, as well as hot loop situations such as mentioned above.

4.2.4.2 Code and Data Decoupling

Normally the code and data segments of a program have a fixed offset in memory, allowing PC-relative addressing of data. However, this also allows an attacker to locate the beginning of the code segment from any global data address. As we describe in Section 4.2.2.6, we decouple the location of the data segment from the code segment, allowing the loader to independently randomize the position of each. To do this, we replace the conventional PC-relative address computation with dynamic relocations assigned by the program loader. This change led to a geometric mean speedup of 4% (labeled *Code and Data Section Decoupling* in Figure 36).

4.2.4.3 Full LR²

The aggregate cost of enabling all techniques in LR² is 6.6% on average (see *Full LR²* in Figure 36). This includes the cost of pointer hiding, software-enforced XoM, register-register addressing restrictions, fine-grained diversity, and the impact of decoupling code and data. This means that our pure software approach to leakage resilient diversity for ARM has about the same overhead as hardware-accelerated leakage resilient diversity for x86 systems (6.6% vs. 6.4% [51]). Because the removal of PC-relative address computations yields a speedup, the cost of individual transformations sometimes exceeds the aggregate cost of LR². An earlier version of our prototype that did not remove PC-relative address computations to decouple code and data sections had an average overhead of 8.4%.

4.2.4.4 Memory Overheads

Finally, in addition to running time, we also measured code section size of the protected SPEC CPU2006 binaries. Forward-pointer hiding had very little overall impact on code size, leading to an increase of 0.9%. Return-address hiding adds at least four instructions to most functions, which resulted in a 5.2% code size increase. The additional load address masking for software-enforced XoM increases the code size by another 10.2%. However, removing the PC-relative address computations decreases the code size by about 14% on average. Comparing the size of full LR² binaries to legacy position independent code shows an average increase of just 5.6%.

4.2.4.5 Impact of XoM-Specific Optimizations

Recall that the differences in threat models between software XoM and SFI (Section 4.2.2.2) allow us to protect multiple uses of a load address using a single masking instruction. To measure the impact of this optimization, we compare the running time of the SPEC CPU2006 benchmarks that run correctly when protected with NaCl to the cost of enforcing XoM. For this experiment, we used the latest version⁶ of the NaCl branch (pnacl-llvm) that is maintained as part of the Chromium project. The results are shown in Figure 37. When enforcing XoM using load masking, the average overhead is 6.6% (for the set of benchmarks compatible with NaCl) whereas

⁶ As of August 10, 2015

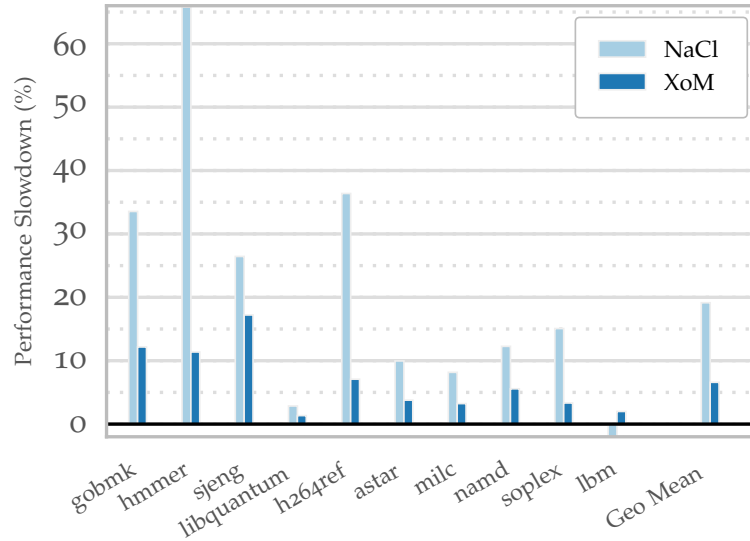


Figure 37: Comparing software XoM to SFI (NaCl) to quantify effect of load-mask optimization.

software-fault isolation, which also masks writes and indirect branches, costs 19.1% overhead. We stress that we are comparing two different techniques with different purposes and threat models. However, these numbers confirm our hypothesis that our XoM-specific load-masking instrumentation reduces overheads. A bigger impact can be seen when comparing code sizes: XoM led to a 5.8% increase, while NaCl caused an increase of 100%. This is a valuable improvement in mobile environments where memory is a scarce resource.

4.2.5 Security Analysis

Our primary goal in LR² is to prevent disclosure of the code layout, which enables sophisticated attacks [196] against code randomization schemes [119]. By securing the code from disclosure we can then rely on the security properties of undisclosed, randomized code.

In order to launch a code-reuse attack the attacker must know the code layout. By applying fine-grained randomization, e.g., function or page reordering, we prevent all static code-reuse attacks, since these attacks are not adjusted to each target’s randomized code layout. For our proof-of-concept we chose to build on function permutation as it is effective, efficient, and easy to implement. However, as all code randomization techniques, function permutation by itself is vulnerable to an attacker who discloses the code layout at run time [45, 61, 196]. Hence, we focus our security analysis of LR² on resilience against direct and indirect information-disclosure attacks targeting randomized program code.

4.2.5.1 *Direct Memory Disclosure*

Direct memory disclosure is when the attacker reads the memory storing randomized code. JIT-ROP [196] is a prominent example of this type of attack. JIT-ROP recursively discloses and disassembles code pages at run time until enough gadgets are disclosed to assemble and launch a ROP attack.

We prevent all direct disclosure attacks by masking memory loads in the protected application, i.e., we prevent loads from reading the program code directly. Masking the load address restricts any attempt to read the code section to the lower half of the memory space which contains only data. Naively masking every load operation is inefficient; we therefore apply the optimizations described in Section 4.2.2.2 to reduce the number of masking instructions. Allowing some unmasked load operations may appear to increase the risk of an unsafe load instruction. However, we are careful to ensure that all unsafe loads are restricted, as we show in the following.

PC-Relative Loads

All PC-relative loads with a constant offset are guaranteed to be safe, since an attacker cannot influence the address used during the load operation and only legitimate data values are loaded in this manner. Therefore, we need not mask these load instructions.

Constant Offsets

We allow loads from known safe base addresses (i.e., already masked values) plus or minus a *small* constant offset (less than 4KiB). Thus, if we ensure that the base address must point into the data section, adding a guard page between the data and code sections prevents the computed address from reaching into the code section. We place an unmapped 8KiB (2 pages) *guard region* between the data and code sections to safeguard all possible constant offsets. In addition, the addresses above 0xC0000000 are reserved for kernel usage and will trigger a fault when accessed, so programs are already safe from address under-runs attempting to read from the highest pages in memory by subtracting a constant from a small base address.

We also allow limited modification of masked addresses without re-masking the result. If an address has already been masked so that it is guaranteed to point into the data section, adding or subtracting a small constant will result in either an address that is still safe, or one that falls into the *guard region*. In either case, the modified address still cannot fall into the code section, and thus we do not need to re-mask it. We perform this optimization for all constant stack pointer adjustments.

Spilled Registers

When a program needs to store more values than will fit into the available registers, it stores (spills) a value temporarily onto the stack to free a machine register. As recently demonstrated, stack spills of sensitive register contents can allow adversaries to completely bypass code-reuse mitigations [45]. In our case, an attacker could attempt to bypass LR² by manipulating a previously masked register while it is spilled to writable

memory. Therefore, we do not trust any address that is restored from writable memory and always re-mask it before the value is used to address memory.

4.2.5.2 *Indirect Memory Disclosure*

Mitigating direct memory disclosure alone does not fully prevent an attacker from leaking the code layout. An attacker can indirectly gain information about the code layout by leaking readable code pointers from the data section [45, 61]. The necessary number of leaked code pointers for a successful code-reuse attack depends on the granularity of the applied randomization. For instance, in the presence of page-based randomization, one code pointer allows the attacker to infer 4 KiB of code due to page alignment, whereas the attacker has to leak more code pointers in the presence of function-level randomization to infer the same amount of code. To counter indirect memory disclosure, we create trampolines for forward code pointers and encrypt return addresses.

Forward-Pointer Protection

An attacker cannot use function pointers to infer the code layout because they point to trampolines which reside in code segment. Hence, the destination address of a trampoline cannot be disclosed. The order of the trampolines is randomized to prevent any correlation between the trampolines and their target functions. This constraints the attacker to whole-function reuse attacks. To mitigate such attacks, we suggest using the XoM-based technique presented by Crane et al. [52] to randomize tables of function pointers. This extension should be completely compatible with the software-only XoM provided by LR² without modification and would protect against the most prevalent types of whole-function reuse: return-into-PLT and vtable-reuse attacks.

Return-Address Protection

Return addresses are a particularly valuable target for attackers because they are plentiful, easy to access, and useful for code-reuse attacks, even with some mitigations in place. For example, when attacking an application protected by function permutation, an attacker can leak return addresses to infer the address of the functions and in turn the addresses of gadgets within those functions [45]. We prevent this by encrypting each return address with a per-function 32-bit random number generated by a secure random number generator. However, our threat model allows the attacker to leak all encrypted return addresses spilled to the stack. While she cannot infer code addresses from the encrypted return addresses we conservatively assume that she can relate each return address to its corresponding call site.

We must also address reuse of unmodified, disclosed return addresses. In a previous indirect disclosure protection scheme, Readactor [51], return addresses were vulnerable to reuse as-is. Although Readactor prevented attackers from gaining information about the real location of code surrounding a call site, an attacker could potentially reuse call-preceded gadgets. An attacker could disclose the trampoline return address

corresponding to a given call site and jump into that trampoline, which in turn jumps directly after the real call site. This allows attackers to reuse any disclosed return addresses. To mitigate this threat, the Readactor authors proposed additional randomizations (register and callee stack slot permutation) to attempt to disrupt data flow between call-proceeded gadgets and mitigate this threat.

In LR² arbitrary reuse of return addresses is impossible. By encrypting every return address with a per-callee encryption key, our system prevents the attacker from invoking a call-site gadget from anywhere but the corresponding callee's return instruction. In other words, encrypted return addresses can only be used to return from the function that originally encrypted the address. Thus, the attacker is confined to the correct, static control-flow graph of the program. This restriction is similar to static CFI policies. However, we further strengthen LR² by applying register-allocation randomization. During our analysis of state-of-the-art ROP attacks we determined that the success of these attack is highly dependent on the data flows between specific registers. Register randomization will disrupt the attacker's intended data flow between registers and hence, have unforeseen consequences on the control flow which will eventually result in a crash of the application.

While our XOR encryption scheme uses a per-function key, this key is shared across all invocations of a function. That is, each time a return address is spilled from a function F it is encrypted with the same key K_F . In most cases this is not a problem, since function permutation prevents an attacker from correlating return addresses encrypted with the same key. However, if a function F_1 contains two different calls to another function F_2 , the return addresses, R_1 and R_2 respectively, are encrypted with the same key K_{F_2} . The attacker has a priori knowledge about these addresses, since with function permutation they are still placed a known (constant) offset apart. We believe this knowledge could be exploited to leak some bits of the key K_{F_2} . To prevent this known-plaintext attack we propose two options: (1) we can either apply more fine-grained code randomization, e.g., basic-block permutation to remove the correlation between return addresses or (2) fall back to using the trampoline approach to protect return addresses as presented by [51] when a function contains more than one call to the same (other) function. These techniques remove the a priori knowledge about the encrypted return addresses. In fact, return-address encryption even strengthens the second approach because it prevents trampoline-reuse attacks for return addresses.

4.2.5.3 Proof-of-Concept Example Exploit

We evaluate the effectiveness of LR² against real-world attacks by re-introducing a known security vulnerability (CVE-2014-1705) into the latest version of Chromium (v46.0.2485.0) and conducted our experiments on same setup we used in our performance evaluation. The vulnerability allows to overwrite the length field of a buffer object. Once this is done we can exploit this manipulated buffer object via JavaScript to read and write arbitrary memory.

We constructed a JIT-ROP style attack that first leaks the vtable pointer of an object O_{target} to disclose its vtable function pointers. Using one of these function pointers we can infer the base address of the code section of Chromium. Next, we use our

1	ldr	r0, [r1, #0]	1	ldr	r0, [r1, #0]
2			2	bic	r0, r0, #0x80000000
3	mov	r12, #28	3	mov	r12, #28
4	ldr	r3, [r0, #7]	4	ldr	r3, [r0, #7]
5	ldr	r1, [r0, #11]	5	ldr	r1, [r0, #11]
6	bfi	r0, r12, #0, #20	6	bfi	r0, r12, #0, #20
7			7	bic	r0, r0, #0x80000000
8	add	r1, r3	8	add	r1, r3
9	ldr	r0, [r0, #0]	9	ldr	r0, [r0, #0]
10			10	add	r1, r1, r2, lsl #2
11			11	bic	r1, r1, #0x80000000
12	ldr	r1, [r1, r2, lsl #2]	12	ldr	r1, [r1]
13	[...]		13	[...]	

Before Instrumentation

After Instrumentation

Figure 38: Simplified disassembly of the function `v8::internal::ElementsAccessorBase::Get` that is used to read arbitrary memory. The load instruction in line 12 reads the memory from the base address provided in register r1 plus the offset in register r2. After the instrumentation, this load is restricted by masking the MSB (line 11) which prevents reads into the code segment.

information disclosure vulnerability to search the executable code at run time for predefined gadgets that allow us to launch a ROP attack to mark data memory that contains our shellcode as executable. Finally, we overwrite the vtable pointer of O_{target} with a pointer to an injected vtable and call a virtual function of O_{target} which redirects control flow to the beginning of our shellcode to achieve arbitrary code execution.

There are currently some efforts by the Chromium community to achieve compatibility with the `musl` C library. By the time of writing this chapter Chromium remains incompatible which prevents us from applying the *full* LR² toolchain. However, we applied our load-masking component while compiling Chromium and analyze the effect this load-masking would have on the memory disclosure we exploit.

Our analysis indicates that Chromium would immediately crash when the attempted code read was restricted into an unmapped memory area within the data section. Figure 38 shows how the function that this exploit uses to leak memory is instrumented. After instrumentation, all load instructions in the function cannot read arbitrary memory and must only read from addresses that point into the data segment. Thus, our proof-of-concept exploit would fail to disclose the code segment at all and would instead crash the browser with a segmentation violation.

4.2.6 Discussion and Extensions

4.2.6.1 Auto-Vectorization

When loops have no data dependencies across iterations, consecutive loop iterations may be vectorized automatically by the compiler, using an optimization technique called

auto-vectorization. This technique computes multiple loop iterations in parallel using vector instructions that perform the same operation on a contiguous set of values.

While investigating the source of the higher overhead for the *hmmr* benchmark, we found that one function—*P7Viterbi*—accounts for over 90% of the benchmark’s execution time. The main loop of this function is amenable to vectorization as it exhibits a high degree of data parallelism [174]. Modern ARM processors support the NEON instruction set extension which operate on four scalar values at a time. Unfortunately, support for automatic vectorization in LLVM was only added in October 2012 and is still maturing. Using the older and more capable vectorization passes in GCC, ICC from Intel, and XLC from IBM may allow more loops to be vectorized [129].

In the context of LR², vectorization would not only reduce the running time by exploiting the data parallelism inherent to many computations; it would also reduce the number of required load masking operations by a factor of more than four. First of all, vectorized loads read four consecutive scalars into vector registers using a single (masked) address. Second, the NEON instructions operate on dedicated, 128-bit wide registers which means that fewer addresses would be spilled to the stack and re-masked when reloaded.

4.2.6.2 *Assembly code*

LLVM does not process inline assembly on an instruction level and therefore transformation passes can only work with inline assembly blocks as a whole. Therefore, our current prototype does not handle inline assembly; this is not a fundamental limitation of our approach however. To make sure that every load is properly masked in the presence of assembly code, we could extend the LLVM code emitter or an assembly-rewriting framework such as MAO [102] with load-masking and code pointer hiding passes. Since the code is not in SSA form at this stage we cannot apply our optimizations.

4.2.6.3 *Dynamically Generated Code*

JIT-ROP attacks are ideally mounted against browsers containing scripting engines. To ensure complete leakage-resilience, we must ensure that XoM and code-pointer hiding is also applied to just-in-time compiled code. Crane et al. [51] patched the V8 JavaScript engine used in the Chrome browser to make it compatible with XoM. To use this patch for LR², we would have to add functionality to ensure that every load emitted by the JIT compiler is properly masked. This would simply involve engineering effort to patch the JIT compiler.

A special property of JIT-compiled code is that it is treated as both code and data by the JIT compiler; when the compiler needs to rewrite or garbage collect the code, it is treated as read-write data, and while running it must be executable. When XoM is enforced natively by the hardware, the page permissions of JIT compiled code can be changed by updating the page tables used by the memory management unit. With software-enforced XoM, we can make JIT compiled code readable by copying it (in part or whole) into the memory range that is accessible to masked loads. However,

that would require a special `memcpy` function containing unmasked loads. Therefore, we believe that a better solution would be to adopt the split-process technique presented by Song et al. [199]. The key idea of this work is to move the activities of the JIT compiler into a separate, trusted process. Specifically, the code generation, optimization, and garbage collection steps are moved to a separate process in which the JIT code cache always has read-write permissions. In the main process, the JIT code cache is always mapped with execute-only (or read-execute if XoM is unavailable) permissions. The two processes access the JIT code cache through shared memory. The main process invokes JIT compilation functionality in the trusted process through remote procedure calls.

4.2.6.4 *Whole-Function reuse attacks*

Since LR² raises the bar significantly for ROP attacks against mobile architectures, attackers may turn to whole-function reuse techniques such as the classic return-into-libc (RILC) technique [151] or the recent counterfeit object-oriented programming (COOP) attack [186]. Our core techniques—execute-only memory and code-pointer hiding—can be extended to mitigate RILC and COOP attacks, as proposed by Crane et al. [52]. To thwart COOP, we would split C++ vtables into a data part (rvtable) and a code part (xvtable) stored on execute-only pages. The xvtable contains trampolines, each of which replaces a pointer to a virtual function. Randomly permuting the layout of the xvtable breaks COOP attacks because they require knowledge of the vtable layout. We can break RILC attacks by similarly randomizing the procedure linkage table (PLT) or analogous data structures in Windows.

4.2.6.5 *Compatibility*

Due to the nature of its load masking and return-address hiding scheme, LR² is fully compatible with unprotected third-party libraries. However, if an unprotected library contains an exploitable memory-disclosure vulnerability it compromises the security of the entire process.

In some cases application developers use the `mmap()` function to map memory to a specific address. In LR² we do not allow mapping to arbitrary addresses because the application will fail when trying to read memory mapped into the XoM region. Hence, we only allow mapping memory into the data region. This is still consistent with the correct semantics of `mmap()` because the kernel considers requested addresses merely as a hint rather than a requirement

4.2.6.6 *AArch64*

Our implementation currently targets 32-bit ARMv7 processors. ARM recently released ARMv8, which implements the new AArch64 instruction set for 64-bit processing. LR² can be ported directly to AArch64. Though AArch64 does not provide a bit clear instruction with immediate operands, we can achieve the same effect with a bitwise AND instruction.

4.2.7 Conclusion

Software that is vulnerable to memory corruption remains exposed to sophisticated code-reuse exploits. The problem of code reuse is not specific to x86 systems but threatens RISC-based mobile and embedded systems too. Code randomization can greatly improve resilience to code reuse as long as the code layout is not disclosed ex post facto. The combination of execute-only memory and code-pointer hiding provides comprehensive resilience against leakage of code layout information. Unfortunately, the implementation of these techniques has so far relied on x86-specific features or has increased resource requirements beyond reasonable limits for mobile and embedded devices.

Unlike previous solutions, our leakage-resilient layout randomization approach—LR²—only requires that the host system enforces a $W \oplus X$ policy. Our software enforcement of execute-only memory is inspired by prior work on software-fault isolation. However, since our threat model is fundamentally different from SFI (we protect trusted code whereas SFI isolates untrusted code), we are able to insert fewer load-masking operations than comparable SFI implementations. This significantly reduces overheads.

We reuse existing techniques to protect forward pointers but present a new optimized XOR pointer encryption scheme relying on XoM and function permutation to protect return addresses. Since LR² does not require any special hardware support, it can protect applications running on a broad range of non-x86 devices, including MMU-less micro-controllers. Even though LR² prevents memory disclosure purely in software, its performance is similar to defenses offering comparable security.

4.3 SELFRANDO: PRACTICAL LOAD-TIME RANDOMIZATION

In the previous sections, we presented techniques to increase the resilience of code-randomization schemes against memory disclosure attacks. For our proof-of-concept implementation of Readactor (Section 4.1) and LR² (Section 4.2) we used compile-time randomization. However, this comes with the disadvantage that the hardened application must be recompiled for every user. In this section, we tackle this issue by designing and implementing practical load-time randomization, coined selfrando. To prove practicability, we enable load-time randomization for Tor Browser [212] which is a browser by the Tor Project for easy access to the Tor Network.

Summing up, our main contributions are:

- **Practical Randomization Framework.** Unlike other solutions that have only been tested on benchmarks, selfrando can be applied to the Tor Browser (TB) without any changes to the source code. To the best of our knowledge, selfrando is the first approach that avoids risky binary rewriting or the need to use a custom compiler, and instead works with existing build tools.
- **Hardening the Tor Browser.** We demonstrate the practicality of selfrando by applying it to the entire TB without requiring any code changes. Our detailed and careful evaluation shows that the startup and performance overheads of selfrando are negligible.

4.3.1 Design and Implementation

Objectives

Our main objective is to design and implement practical and efficient load-time randomization that can be combined with existing memory-disclosure mitigations. For practicality reasons, we choose to support complex C/C++ programs (e.g., a browser) without modifying their source code. Further, we retain full compatibility with current build systems, i.e., we should avoid any modification to compilers, linkers, and other operating system components. To be applicable for privacy-preserving open-source tools, we must not rely on any third-party proprietary software. Finally, our solution should not substantially increase the size of the program in memory or on disk.

Selfrando

The easiest way to perform fine-grained code randomization is by customizing the compiler to take a seed value and generate a randomized binary [71, 96]. Unfortunately, compiling and distributing a unique binary for each is impractical for introducing diversity among a population of programs [70, 230]. With more implementation effort, we can delay randomization until load-time, which has several benefits. Most importantly, software vendors only need to compile and test a single binary. A single binary also means that users can continue to use hashes to verify the authenticity of

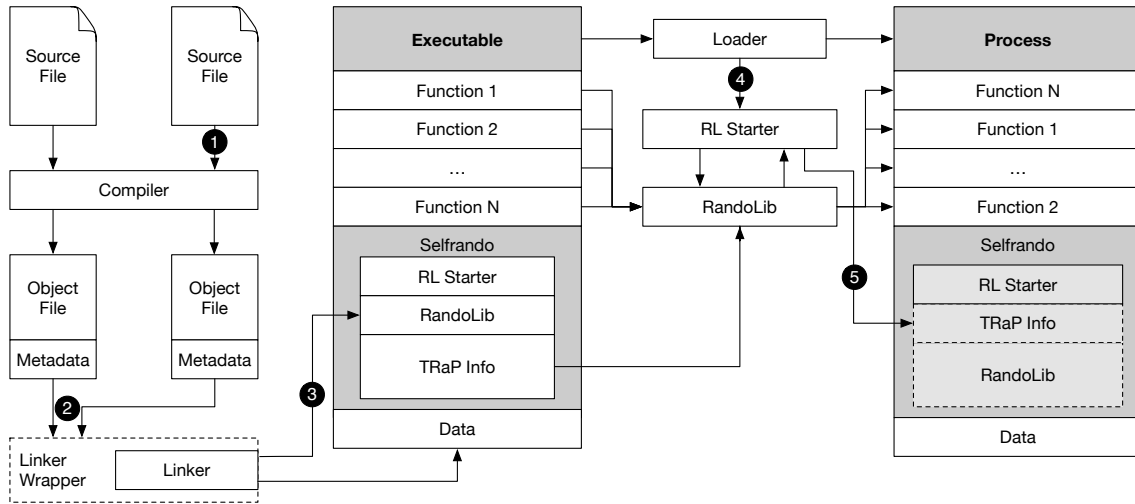


Figure 39: Workflow of selfrando.

the downloaded binary. Finally, modern content delivery networks rely extensively on caching binaries on servers; this optimization is no longer possible with unique binaries.

In the context of privacy-preserving software such as TB, compile-time randomization raises additional challenges. Randomized builds would complicate the deterministic build process⁷, which is important to increase trust in the distributed binary. Moreover, compile-time randomization would (a) increase the feasibility of a de-anonymization attack due to individual, observable characteristics of a particular build, and (b) allow an attacker to build knowledge of the memory layout across application restarts, since the layout would be fixed.

For these reasons, we decided to develop a framework which makes the program binary randomize itself at load time. We chose function permutation [113] as the randomization granularity, since it dramatically increases the entropy of the code layout while imposing the same low overheads as ASLR [218]. Since discovering function boundaries at load-time by analyzing the program binary is unreliable and does not scale to large programs, we pre-compute these boundaries statically and store the necessary information in each binary. We call this *Translation and Protection* (TRaP) information.

Rather than modifying the compiler or linker, we developed a small tool which wraps the system linker, extracts all function boundaries from the object files used to build the binary, then appends the necessary TRaP information to the binary itself. Our linker wrapper works with the standard compiler toolchains on Linux and Windows and only requires a few changes to the build scripts to use with the TB.

Figure 39 illustrates the overall design and workflow of selfrando. First, the unmodified source code is compiled into object files ❶. Object files are comprised

⁷ A randomized build can be implemented in a deterministic environment by passing a random seed as an input to the deterministic process. The builds would then be distributed along with their seed. A user could then check the integrity of her build by running the deterministic process again with the same seed. However, that check would not prove the integrity of builds with other seeds.

of the compiled binary code of the respective source file, as well as, metadata, like function boundaries and references, which are required by the linker to combine object files into the one executable file. In a normal compilation process these metadata are not included by the linker in the final binary. However, our linker wrapper extracts all metadata, which are required to re-randomize the binary during load time ❷. The extracted information of each object file are bundled into the TRaP information, and embedded together with a load-time randomization library, RandoLib, into the binary file ❸.

Pre-compiled language runtime object files are another obstacle. One example is `crtbegin.o` for GCC which contains functions to initialize the runtime environment for applications that were programmed in C. In our current implementation, we treat such object files as one single block because they contain only a few functions. This has a negligible impact on the overall randomization entropy.

When the loader loads the application, it will invoke RandoLib instead of the entry point of the application. RandoLib performs function permutation using the embedded TRaP info, and consists of two parts: a small helper stub and the main randomization module. The purpose of the helper stub (*RL Starter*) is to make all selfrando data inaccessible after RandoLib finishes. The operating system loader ❹ calls this stub, invoking RandoLib as the first step of program execution. The function permutation algorithm proceeds in several steps. First, RandoLib generates a random order for the functions using the Fisher-Yates shuffling algorithm. Second, RandoLib uses the embedded metadata to fix all references that became invalid during the randomization. Finally, after RandoLib returns, the helper stub makes selfrando's data inaccessible ❺, and jumps to the original entry point of the binary.

4.3.2 Evaluation

We thoroughly evaluated selfrando from a security, and load- and run-time performance standpoint.

Security

For any randomization scheme the amount of entropy provided is critical, because a low randomization entropy enables an attacker to guess the randomization secret with high probability [193]. We compare selfrando to Address Space Layout Randomization (ASLR)—the standard code randomization technique that is available on all modern systems.

We determined the real-world entropy of ASLR by running a simple, position-independent program multiple times and analyzing the addresses, on a Debian 8.4 machine using GCC 6.1.0 and Clang 3.5.0. ASLR provides up to 9-Bits of entropy on 32-bit systems and up to 29-Bits of entropy on 64-bit systems. While the ASLR offset on 32-bit systems is guessable in a reasonable amount of time, such attacks become infeasible on 64-bit systems because the address space is that much larger. However, an attacker can bypass ASLR by leaking the offset that the code is loaded at in memory

through a pointer into application memory. Once this offset is known the attacker can infer any address within the application, because it is used to shift the address of the whole application.

selfrando, on the other hand, applies more fine-grained function permutation. This means the randomization entropy does not depend on the size of the address space, as it is the case for ASLR, but on the number of functions in the randomized binary.

We applied selfrando to Tor Browser, and analyzed the entropy for different library. The smallest library (`libp1ds4.so`) has 44 functions in 10 KB of code, while the biggest (`libxul.so`) has 242 873 functions in 92 MB. The median is 494 functions in 163 KB, while the average is 16 814 functions in 6.5 MB. With the assumption that the attacker needs the address of at least three functions, selfrando is significantly more effective than ASLR. For the smallest library, the attacker needs to guess at least 39-Bits, while for the biggest, the attacker needs at least 78-Bits.

To protect selfrando against sophisticated memory-disclosure attacks it must be combined with some form memory-disclosure mitigation, like Readactor (Section 4.1) and LR² (Section 4.2).

Run-time Performance

We performed multiple tests to measure selfrando's run-time overhead. Since selfrando works at load-time, we also measured the additional startup time. All tests were performed on a system with an Intel Core i7-2600 CPU clocked at 3.40 GHz, with 12 GB of RAM and a 7200 RPM hard disk. We used version 5.0.3 of the Tor Browser on Ubuntu 14.04.3.

We executed all the C and C++ benchmarks in SPEC CPU2006 with the two standard Linux compilers (GCC and Clang) with selfrando enabled. Moreover, we ran the benchmarks with a version of selfrando that always chooses the original order for the randomization (*identity transformation*). This version runs all the load-time code but it does not actually modify the code segment. It allows us to distinguish between load-time overhead and run-time overhead.

The geometric mean of the positive overheads is 0.71% for GCC and 0.37% for Clang. We found one of the benchmarks programs, `xalancbmk`, to be an outlier, with an overhead of about 14%.

We investigated this issue using the Linux performance analysis tool, `perf`, comparing the full selfrando and the identity transformation runs. We discovered a 69% increase in L1 instruction cache misses and a 521% increase in instruction TLB (Translation Lookaside Buffer) misses. We believe that the `xalancbmk` benchmark is sensitive to the function layout and that some frequently executed functions must be co-located to ensure optimal performance. A possible extension to selfrando to cope with location-sensitive programs is to automatically use performance profiling to identify groups of functions that should be moved as a single bundle similar to the work of Homescu et al. [95].

Load-time Performance

Finally, we evaluated the load-time overhead using the standard tool time. As a baseline, we used the source code of Tor Browser 5.0.3, unmodified except for the main function. For both versions, the reported time is the average of 10 runs. We cleaned the disk cache before each run, so the binary was loaded from the disk every time.

The average load time for the normal version was 2.046 s, while the selfrando version took 2.400 s on average. The average overhead is 354 ms. We believe this is an acceptable overhead.

4.3.3 Conclusion

We have introduced selfrando, a fast and practical load-time randomization tool. It has negligible run-time overhead, a perfectly acceptable load-time overhead, and it requires no changes to the source code.

We successfully tested selfrando with a variety of different software. Further, in a collaboration with the Tor Project we integrated selfrando into Tor Browser which is currently distributed in the beta version for Linux users.

Moreover, selfrando can be combined with integrity techniques such as execute-only memory to further secure the Tor Browser and virtually any other C/C++ application.

4.4 PT-RAND: MITIGATING ATTACKS AGAINST PAGE TABLES

Operating system kernels are essential components in modern computing platforms since they provide the interface between user applications and hardware. They also feature many important services such as memory and disk management. Typically, the kernel is separated from user applications by means of memory protection, i.e., less-privileged user applications can only access the higher-privileged kernel through well-defined interfaces, such as system calls. Attacks against kernels are gaining more and more prominence for two reasons: first, the kernel executes with high privileges, often allowing the attacker to compromise the entire system based on a single kernel exploit. Second, the kernel implements a major part of the security subsystem. Hence, to escalate execution privileges to *root* or escape from application sandboxes in browsers, it is often inevitable to compromise the kernel. Kernel exploits are leveraged in (i) all of the latest iOS jailbreaks, (ii) browser sandbox exploits against Chrome [146], and (iii) large-scale attacks by nation-state adversaries to obtain full control of the targeted system, as in the infamous case of Stuxnet [175].

Typical means for program code exploitation are *memory corruption* vulnerabilities. They allow attackers to alter control and data structures in memory to execute (injected) malicious code, or to launch code-reuse attacks using techniques such as return-oriented programming [100, 192]. One of the main reasons for the prevalence of memory corruption vulnerabilities is that a vast amount of software is programmed in unsafe languages such as C and C++. In particular, kernel code is typically completely written in these languages for better performance, legacy reasons, and hardware-close programming. The monolithic design of the commodity kernels and numerous device drivers increase the attack surface compared to user-mode applications. For instance, over the last 17 years 1526 vulnerabilities have been documented in the Linux kernel [54].

Various solutions have been proposed or deployed in practice to protect software systems against code-injection or code-reuse exploits: modern kernel hardening solutions like *Supervisor Mode Execution Protection* (SMEP) and *Supervisor Mode Access Protection* (SMAP) [106] prevent access to user-mode memory while the CPU executes code in kernel mode [12, 106]. This prevents the attacker from executing code with kernel privileges in user mode. The deployment of $W \oplus X$ (Writable \oplus Executable) prevents the attacker from executing code in the data memory. Indeed, $W \oplus X$ has dramatically reduced the threat of code-injection attacks. However, attackers have already eluded to more sophisticated exploitation techniques such as code reuse to bypass these measures and to hijack the control flow of the targeted code. Mitigating *control-flow hijacking* attacks is currently a hot topic of research [205]. The most promising and effective defenses at the time of writing are control-flow integrity (CFI) [6], fine-grained code randomization [119], and code-pointer integrity (CPI) [118]. However, all defenses against control-flow hijacking are based on the following assumptions: firstly, they assume that code pages cannot be manipulated. Otherwise, the attacker can replace existing code with malicious code or overwrite CFI/CPI checks. Secondly, they assume that critical data structures containing code pointers (e.g., the shadow stack for

CFI, the safe region for CPI) are isolated. Otherwise, the attacker can manipulate them by overwriting code pointers.

However, as observed by Ge et al. [78], defenses against control-flow hijacking in the kernel additionally require the protection of *page tables* against *data-only attacks*. Otherwise the assumptions mentioned above will not hold and these defenses can simply be bypassed by manipulating the page tables.

Data-only attacks do not change the control flow of the program. Instead they direct the control flow to certain nodes within the control-flow graph (CFG) of the underlying program by altering the input data. Hence, the executed path in the CFG is indistinguishable from any other benign execution. Page tables are data structures that map virtual addresses to physical addresses. They define *read-write-execute* permissions for code and data memory pages, where a page is simply a contiguous 4KB memory area. Hence, attackers can launch data-only attacks (based on memory corruption vulnerabilities in the kernel) to alter page tables, and consequently disable memory protection, manipulate code pages, and inject malicious code [147]. Recently industry researchers have presented several page-table based attacks [65] stressing that these attacks are possible because the attacker can easily determine the location of the page tables.

To tackle data-only attacks on page tables, previous work suggested kernel instrumentation to mediate any access to memory-management structures according to a security policy [14, 15, 53, 77, 176, 200]. However, all these solutions suffer from at least one of the following shortcomings: high performance overhead, require additional and higher privileged execution modes (e.g., hypervisors), or depend on architecture-specific hardware features. Recently, Microsoft released a patch for Windows 10 [108] that randomizes the base address used to calculate the virtual address of page table entries. However, this patch does not protect against crucial information disclosure attacks that have been frequently shown to circumvent any (even fine-grained) randomization scheme [61, 196].

Goal and Contributions. In this chapter, we present the design and implementation of a novel memory protection scheme, PT-Rand, that prevents the attacker from manipulating page tables. We highlight the importance of page table protection by implementing a real-world exploit, based on a vulnerable kernel driver (CVE-2013-2595), to directly manipulate the code of a kernel function. Using this attack, we circumvent a recently released CFI kernel hardening scheme, Linux RAP [166], and execute arbitrary code with kernel privileges. In summary, our contributions are as follows:

- **Page Table Protection.** We present a practical and effective protection of page tables against *data-only attacks* without requiring additional hardware or a hypervisor. Rather than applying expensive policy enforcement checks, we randomize page tables when they are allocated and ensure that no information related to the location of page tables is leaked. To achieve this, we need to tackle several challenges. (1) There are many data pointers that the attacker can exploit to locate page tables. (2) The physical memory (including page tables) is usually mapped 1:1 into the virtual address space. Hence, the attacker can easily locate and

access this section. (3) The kernel still needs to efficiently access page tables, and distinguish between randomized and regular memory pages. As we will show in Section 4.4.5.1, PT-Rand tackles all these challenges, while remaining compatible to existing software, like kernel drivers.

- **Prototype Implementation.** We provide a fully working prototype implementation for a recent Linux kernel (v4.6). We also combine Linux kernel CFI protection (RAP) with PT-Rand to protect RAP against data-only attacks on page tables.
- **Performance Evaluation.** We provide an extensive security and performance evaluation. In particular, we show that the attacker cannot bypass the randomization by means of guessing attacks. Our performance measurements for popular benchmarking suites SPEC CPU2006, LMBench, Phoronix, and Chromium browser benchmarks show that PT-Rand incurs almost no measurable overhead (0.22% on average for SPEC), successfully applies to many complex, modern system configurations, and is highly practical as it supports a variety of applications and kernel code.

PT-Rand effectively enables memory protection and paves the way for secure deployment of defenses to thwart code-reuse attacks on the kernel.

4.4.1 Background on Memory Protection and Paging

In this section, we recall the basic principles of memory protection and paging that are needed for the understanding of the following sections.

Memory protection ensures that (i) privileged kernel code is isolated from less-privileged user code, (ii) one process cannot access the memory space of another process, and (iii) read-only data memory cannot be tampered with by unauthorized write operations. To enforce memory protection, modern operating systems leverage a widely-deployed CPU feature called *paging*. Although the implementation details vary among different architectures, the basic principles are the same. Hence, without loss of generality, we focus our discussion on paging for the contemporary x86_64 architecture.

Paging creates an indirection layer to access *physical memory*. Once enabled, the CPU will only operate on *virtual memory* (VM), i.e., it can no longer access physical memory. The advantage of paging is that processes start working with large contiguous memory areas. However, physically, the memory areas are scattered throughout the RAM, or swapped out on hard disk. As a consequence, each access to a virtual memory address needs to be translated to a physical address. This is achieved by a dedicated hardware engine called Memory Management Unit (MMU). The translation is performed by means of *page tables* that operate at the granularity of pages, where a typical page size is 4KB. Specifically, the operating system stores mapping information from virtual to physical addresses into these page tables thereby enabling efficient translation. To isolate processes from each other, the kernel assigns each process to its own set of page tables. In addition, page tables maintain *read-write-execute* permissions for each memory page.

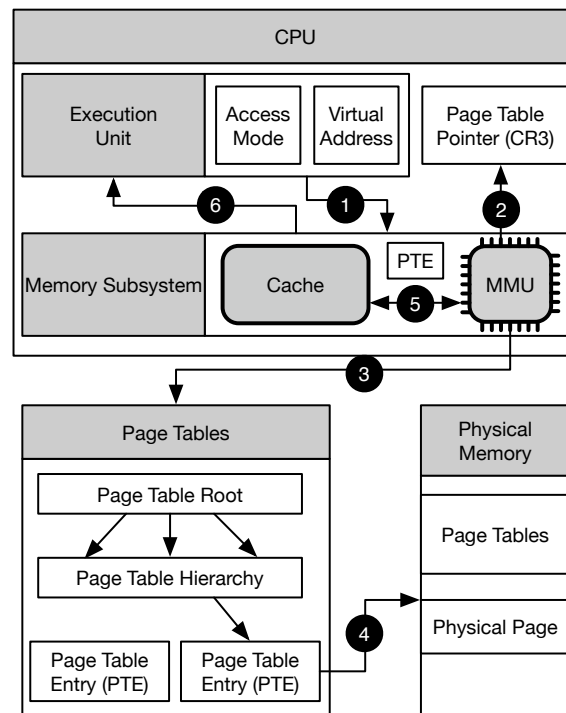


Figure 40: Paging - translation of virtual addresses to physical addresses.

These permissions are enforced at the time of translation, e.g., allowing the operating system to prevent write operations to code pages or executing data pages.

Figure 40 provides high-level insights into the translation process. First, the memory subsystem of the CPU receives the access mode and a virtual memory address from the execution unit as input ❶. To access the page tables, the MMU reads out the pointer to the page table root which is always stored in the third control register (CR3) on x86_64 ❷. This pointer is already a physical memory address pointing to the root of the page table hierarchy ❸. That said, page tables are organized in a tree-like hierarchy for space optimization reasons. The MMU traverses the page table hierarchy until it reaches the page table entry (PTE) which contains the physical address for the given virtual memory address ❹. In addition, the PTE holds the access permissions and ownership (user or kernel) of the associated memory page. The memory subsystem leverages this information to validate whether the target operation (read, write, or execute) adheres to the permission set and ownership of the page. If validation is successful, the translation information is used to fetch the data from the physical memory slot and stored into the cache ❺. Note that the cache internally consists of a data and an instruction cache. For read and write operations the fetched data is stored into the data cache. In contrast, execute requests lead to a write of the fetched data to the instruction cache. Finally, the fetched data is forwarded to the execution unit of the CPU ❻. If the MMU either does not find a valid mapping in the page table hierarchy or observes an unauthorized access in ❹, the memory subsystem generates an exception ❻.

It is important to note that the page tables only contain physical addresses. This becomes a performance bottleneck when the kernel aims at changing the page permissions. As the kernel operates on virtual addresses, all the physical addresses belonging to a page would need to be mapped to virtual addresses dynamically before the permission update can be performed. To tackle this bottleneck, the kernel maintains a so-called *1:1 mapping* which permanently maps the whole physical memory to a fixed address into the virtual memory. To quickly translate a physical to a virtual address, the kernel adds the physical address to the start address of the 1:1 mapping, and can then use the resulting virtual address to access the memory.

4.4.2 On the Necessity of Page Tables Protection

In the adversary setting of kernel exploits the attacker has full control of the user mode, and hence, can execute arbitrary code with user-mode privileges, and interact with the kernel through system calls and driver APIs. The attacker's goal is to gain higher privilege level to be able to execute arbitrary code with kernel-mode privileges. To do so, the attacker needs to hijack a control-flow path of kernel code by overwriting a kernel code pointer, e.g., a return address or function pointer, using a memory-corruption vulnerability that is exposed either through the kernel itself or one of the loaded drivers.

In the following, we briefly provide an overview of the main kernel-related exploitation techniques as well as the defenses that are deployed or proposed against these attacks. To mitigate kernel code-injection and kernel code-reuse attacks, the kernel must be hardened with a variety of protection measures such as $W\oplus X$ and Control-Flow Integrity (CFI), fine-grained randomization or Code-Pointer Integrity (CPI). However, as we elaborate in the following the security of all these defenses relies on the integrity of page tables that can be attacked by means of data-only attacks – We show this using a real-world exploit that manipulates page tables against a kernel CFI protection.

4.4.2.1 Traditional Kernel Attacks

To escalate the attacker's privileges to kernel privileges, a common exploitation technique is as follows: first, the attacker allocates a new buffer in memory, writes malicious code into this buffer, and sets the memory page on which the buffer is located to executable. The latter can be achieved by common user space library functions such as *mprotect()* on Linux and *VirtualProtect()* on Windows. Recall that these actions are possible because the attacker has already gained control of the user space. Second, the attacker overwrites a kernel code pointer with the start address of the malicious code based on a memory corruption vulnerability inside the kernel. These vulnerabilities are typically triggered by abusing the kernel's interfaces such as system calls and driver APIs. Third, the attacker triggers the execution of a function that executes a branch on the corrupted kernel code pointer. As a result, the kernel's internal control flow will be dispatched to the previously injected, malicious code. Although this code resides in user space, it will be executed with kernel privileges because the control-flow hijacking occurred in the kernel mode. In a similar vein, the attacker can launch code-reuse attacks

using the return-oriented programming (ROP) [192] technique. These attacks combine and chain short instruction sequences (called gadgets) that end in an indirect branch instruction. They are typically leveraged if the attacker cannot allocate new malicious code on an executable page. Thus, the user-mode buffer will hold a ROP payload consisting of code pointers to gadgets. Upon corruption of the kernel pointer, the ROP gadget chain will be executed under kernel privileges [67].

4.4.2.2 Code-injection and Code-reuse Attacks

Modern CPUs feature hardware extensions *Supervisor Mode Execution Protection* (SMEP) and *Supervisor Mode Access Protection* (SMAP) that prevent access to user-mode memory while the CPU executes code in the kernel mode [12, 106]. Alternatively, if these extensions are not present, the kernel can simply unmap the entire user space memory when kernel code is executed [130]. Such protections force the attacker to directly inject malicious code or the ROP payload into the kernel's memory space which is a challenging task since the attacker cannot directly write into kernel memory. However, several kernel functions accept and process user-mode buffers. A prominent example is the *msgsnd()* system call which allows exchange of messages. The attacker can exploit this function to cause the kernel to copy the user-mode exploit buffer (the message) into kernel memory. By leveraging a memory disclosure attack inside the kernel, the attacker can determine the address where the buffer is located in kernel memory and launch the exploit thereafter [169]. Several techniques are deployed or proposed to harden the kernel against these attacks: $W \oplus X$ (Writable \oplus Executable) is leveraged by many modern operating systems to prevent code to be executed from data memory. *Fine-grained code randomization* diversifies the code address layout to complicate code-reuse attacks [119]. Many modern operating systems apply Kernel Address Space Layout Randomization (KASLR) [66, 130]. *Control-flow integrity* (CFI) mitigates control-flow hijacking attacks by validating that the application's control flow remains within a statically computed control-flow graph [6]. CFI has been also adapted to kernel code [53, 78]. Recently a CFI-based protection for Linux kernel (RAP [166]) has been released. *Code pointer integrity* (CPI) [118] prevents control-flow hijacking by ensuring the integrity of code pointers, and pointers to code pointers.

Principally all these defenses significantly raise the bar. However, as observed in [53, 78] these defenses heavily rely on the assumption that the instrumented code cannot be manipulated, i.e., the attacker cannot compromise integrity checks or exploit information leakage against randomization schemes, and replace existing code with malicious code. On the other hand, this assumption is easily undermined by data-only attacks that tamper with the page tables as we describe next.

4.4.2.3 Data-only Attacks against Page Tables

In contrast to control-flow hijacking attacks, data-only attacks abstain from compromising code pointers. For example, the attacker can overwrite the *is_admin* variable of an application at run-time [38]. Although no code pointer has been compromised, the attacker can now execute benign functionality with higher privileges.

In the context of the kernel, data-only attacks allow code injection attacks by modifying page table entries (PTEs) which we explained in Section 4.4.1. To initiate data-only attacks, the attacker first exploits a memory-corruption vulnerability in the kernel or a device driver to gain read and write access to kernel memory. Since kernel memory contains references to page tables, the attacker can carefully read those references and locate them [147]. In particular, the attacker can disclose the virtual address of a PTE corresponding to a page that encapsulates a kernel function which can be triggered from the user space. Next, the attacker modifies the page permissions to writable and executable. For instance, the entire code of the kernel function could be replaced with malicious code. Finally, the attacker triggers the kernel function from user space to execute the injected code with kernel privileges.

4.4.2.4 *Generic bypass of Kernel CFI*

To demonstrate the potential of data-only attacks against page tables, we first hardened the current Linux kernel with the open source version of RAP [166]. RAP is a state-of-the-art CFI implementation that instruments the Linux kernel during compile-time to enforce fine-grained CFI at run-time. In particular, RAP ensures that the attacker cannot overwrite code pointers (used for indirect branches) with arbitrary values. This is achieved by emitting CFI checks before all indirect branches that validate whether the program flow targets a valid destination. However, as mentioned before, a fundamental assumption of RAP is the integrity of the kernel code. If code integrity is not ensured, the attacker can simply overwrite the CFI checks with NOP instructions or directly overwrite existing kernel code with malicious code.

We undermine this assumption by using a data-only attack to first modify the page tables and change the memory permission of the kernel code to writable. Next, we overwrite an existing system call with our attack payload which elevates the privileges of the current process to root. After successfully overwriting the kernel code, we invoke the modified system call from user mode to eventually obtain root access. The details of this exploit are described in Section 4.4.5.2. While the impact of the attack itself is not surprising (CFI does not aim to prevent code-injection attacks), it highlights the importance of having an effective protection against data-only attacks that target page tables. We note that this attack is not limited to RAP but can also be applied to randomization or isolation-based defenses (CPI) against code-reuse attacks.

4.4.2.5 *Summary*

All known exploit mitigation schemes strongly depend on memory protection to prevent the attacker from injecting code or corrupting existing code. Even with these schemes in place, page tables managing memory permissions can be compromised through data-only attacks. Hence, designing a defense against data-only attacks is vital and complements the existing mitigation technologies allowing their secure deployment for kernel code.

4.4.3 Threat Model

The adversary setting for our protection scheme PT-Rand against page tables corruption is based on the following assumptions (which are along the lines of the assumptions of related literature):

Defense Capabilities

User Space Access. User-mode pages are not accessible when the CPU is in the kernel mode. This is enforced by modern CPU features such as SMAP/SMEP [12, 106] or by simply unmapping the user space during kernel code execution [130].

Writable \oplus Executable. Kernel code pages are not per-se writable. This is enforced by W \oplus X protection inside the kernel. As a consequence, the attacker needs to resort to a **data-only attack** to manipulate code page permissions, and inject code thereafter.

Code-reuse Defense. A defense mechanism against kernel-related code-reuse attacks is enforced, such as control-flow integrity (CFI) [6, 78], fine-grained code randomization [51, 119], or code-pointer integrity (CPI) [118]. Specifically, our prototype implementation of PT-Rand incorporates RAP [166], a public state-of-the-art CFI implementation for the Linux kernel. As mentioned before, existing defenses against code-reuse attacks cannot prevent data-only attacks against the page tables. (Our solution serves as a building block to prevent these protection frameworks from being undermined by data-only attacks against page tables.)

DMA Protection. Direct Memory Access (DMA) [181, 231] cannot be exploited to bypass virtual memory permissions because an IOMMU [106] is configured to prevent DMA to security-critical memory.

Safe Initialization. The attacker cannot attack the kernel prior the initialization of PT-Rand. This is not a limitation because PT-Rand is initialized at the early boot phase during which the attacker cannot interact with the kernel.

Source of randomness. A secure (hardware) random number generator is available [12, 106, 217].

Side-channels. Timing and cache side channel attacks as well as hardware attacks, like rowhammer [114], are orthogonal problems, and hence, beyond the scope of this chapter. Nevertheless, we discuss in Section 4.4.5.2 how we can adopt known techniques from Apple's iOS to prevent practical side-channel attacks.

Adversary Capabilities

Memory Corruption. There exists a memory corruption vulnerability in either the kernel or a driver. The attacker can exploit this vulnerability to read and write arbitrary memory (e.g., [147]).

Controlling User Space. The attacker has full control of the user space, and consequently can execute arbitrary code in user space and call kernel API functions.

4.4.4 Overview of PT-Rand

Our goal is to mitigate data-only attacks against the kernel page tables in the threat model explained in section 4.4.3. To do so, we introduce the design and implementation of a novel kernel extension called PT-Rand. The main idea of PT-Rand is to (i) *randomize* the location of page tables securely, i.e., *prevent* the leakage of the randomization secret, and (ii) *substitute* pointers that reference page tables with physical addresses to obfuscate these references and prevent their leakage.

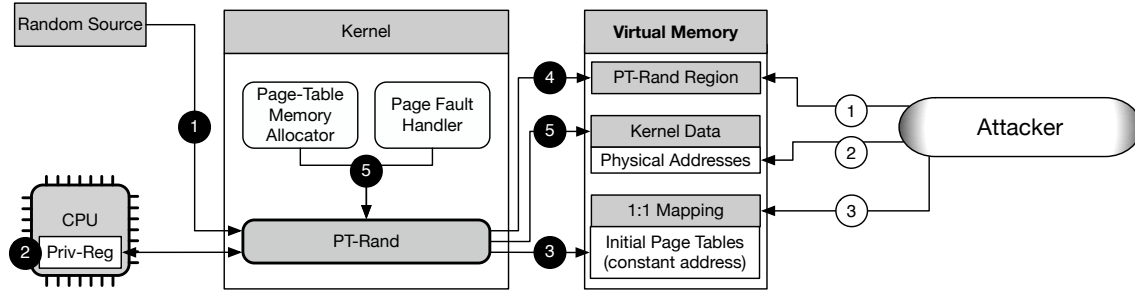


Figure 41: Overview of the different components of PT-Rand.

Figure 41 depicts the overall architecture and workflow of PT-Rand. During the early boot phase, the kernel operates only on physical memory. To guarantee a successful switch to virtual memory, contemporary kernels allocate an initial set of page tables at a constant and fixed address. These page tables manage the kernel’s core functions as well as data areas, and remain valid for the rest of the kernel’s life-time. To prevent the attacker from tampering with page tables, PT-Rand generates a randomization secret ❶, and randomizes the location of the initial page tables ❷. The randomization secret is stored in a privileged CPU register which is neither used during normal operation of the kernel nor accessible from user mode. Recall from Section 4.4.3 that the attacker can only access the kernel memory, but not the kernel’s registers. The latter would require the attacker to either launch a code-injection attack (prevented by $W \oplus X$) or a code-reuse attack (mitigated by CFI [6], code randomization [119] or CPI [118]). After relocating the initial page tables to a random address, the kernel can no longer access these page tables through the 1:1 mapping. In particular, PT-Rand relocates the initial page tables in an unused memory region. As we will evaluate in detail in Section 4.4.5.2, the entropy for this memory region is reasonably high for contemporary 64-bit systems rendering brute-force attacks infeasible ㉞.

Note that the kernel features dedicated allocator functions for page table memory. For PT-Rand, we instrument these functions to (i) move the initial page tables to a random address, and (ii) always return physical addresses for any page table related memory allocation. In contrast, the default allocators always return a virtual address as a reference to newly allocated page table memory. This small adjustment allows us to obfuscate the location of page tables from user-level attackers, because the kernel code operates on virtual addresses when accessing page tables. Hence, at this stage, neither the attacker nor the kernel itself can access the page tables. In order to allow benign

kernel code to still access the page tables, we modify all kernel functions that access page table memory: for each of these functions we convert the physical address to a virtual address based on the randomization secret generated in ❶.

However, during the early boot phase, the kernel has already saved references to the initial page tables in various data structures. Since the initial tables were not allocated with our modified allocator, the references contain obsolete virtual addresses. To avoid a kernel crash, PT-Rand updates all these references (virtual addresses) with the new physical address ❸. To this end, every reference to page tables now contains a physical address rather than a virtual address. Thus, the attacker aiming to locate page tables by reading the designated places of page table pointers [147] only retrieves physical addresses. Since there is no direct correlation between physical and virtual addresses, the attacker cannot use any leaked references to infer the corresponding virtual address ❹. We also implemented PT-Rand such that no intermediate computation result that includes the randomization secret is ever written into memory. Specifically, we instruct the compiler to keep intermediate and the end result that include the randomization secret in registers, and prevent them from getting spilled.

Our modified page table memory allocator also randomizes any future page table allocations into the PT-Rand memory region ❺. Further, we ensure that every physical memory page that contains page table entries is unmapped from the 1:1 mapping. Hence, if the attacker discloses a physical address of a page table pointer, she cannot exploit the 1:1 mapping to read out page tables ❻. Finally, PT-Rand provides an interface for the kernel to access and manage page tables ❼. In particular, PT-Rand translates the physical addresses of page table pointers to virtual addresses based on the randomization offset.

4.4.5 *Implementation and Evaluation*

In this section, we present the implementation and evaluation results for PT-Rand. For our evaluation, we first analyze security aspects such as randomization entropy and leakage resilience. Thereafter, we present a thorough investigation of the performance overhead incurred by PT-Rand. For this, we conducted micro-benchmarks on hot code paths, measure performance overhead based on SPEC CPU industry benchmarks, and quantify the impact on complex applications such as browsers.

4.4.5.1 *Implementation*

We implement PT-Rand for the Linux kernel version 4.6. However, the concepts of PT-Rand can be applied to other kernels as well. Our modifications mainly target the memory allocator for page tables, and the part of the page fault handler that is responsible for traversing the page tables. Further, we extend the initialization code of the kernel to relocate page tables, which were required during the boot time.

To provide sufficient entropy, PT-Rand requires a large memory area to store the page tables. Therefore, we first analyzed the usage of the 64-bit address space of the Linux kernel which is shown in Figure 42. We identified two unused regions of 40 bit

Address	Size	Purpose
0x000000000000	47 Bits	User Space
hole caused by [48:63] sign extension		
0xfffff8000000	43 Bits	Hypervisor
0xfffff8800000	43 Bits	1:1 Mapping
0xfffffc800000	40 Bits	PT-Rand (Hole)
0xfffffc900000	45 Bits	vmalloc/ioremap
0xfffffe900000	40 Bits	Hole
0xfffffea00000	40 Bits	Memory Map
unused hole		
0xfffffec00000	44 Bits	Kasan
unused hole		
0xfffffff00000	39 Bits	Fixup Stacks
unused hole		
0xfffffffff800	512M	Kernel Text
0xfffffffffa00	1525M	Modules
0xfffffffffff6	8M	vsyscalls
0xffffffffffe	2M	Hole

Figure 42: The x86_64 virtual memory map for Linux with four level page tables.

which each translates to one TB of memory, and we utilize one of them to store the page tables at a random address. To randomize the location of the page tables, we first generate a random address within this region using the available kernel Application Programming Interface (API), which uses a hardware random number generator if available. PT-Rand stores this randomization secret in the DR3 register, which is normally used for debugging purposes. We modify the page fault handler to use this secret to traverse the page tables, as well as the memory allocator for page tables to use the register to obfuscate the page table addresses. In particular, we modify the allocator such that all pointers to the page table are physical addresses, and can only be converted into virtual addresses with the randomization secret that is stored in the DR3 register. Next, we relocate the page tables that were used during boot time to set up the initial virtual address space. This is necessary because these page tables are statically included in the binary. Finally, PT-Rand ensures that the memory, in which the page tables are stored, is not accessible through the 1:1 mapping which maps the entire physical memory.

4.4.5.2 Security Considerations

Our main goal is to prevent data-only attacks against the kernel page tables at run time. For this, we randomize the location of page tables per boot. In general, any randomization-based scheme must resist the following attack vectors: (i) guessing attacks, (ii) memory disclosure through code and data pointers, and (iii) memory disclosure through spilled registers. In the following, we discuss each attack vector to demonstrate the effectiveness of PT-Rand. We also include an exploit in our study

to demonstrate that exploit hardening mechanisms at the kernel-level can be bypassed when PT-Rand is not applied.

Guessing Attacks

Low randomization entropy allows the attacker to guess the randomization secret with high probability [193]. The randomization entropy of PT-Rand depends on: (1) the number of guesses, (2) the size of the region where the page tables are allocated, and (3) the overall size of memory that is required to store all page tables.

We limit the number of attacker's guesses by configuring the kernel to initiate a shutdown in case of an invalid memory access in kernel memory. Note that this has no impact on the kernel's execution. In fact, this was the default behavior of previous versions of the Linux kernel. As described in Section 4.4.5.1, we utilize an unused memory region of 1TB (40 Bit) to randomize the memory allocations for the page tables. However, the smallest memory unit in paging is a 4KB (12 Bit) page. This means when one page table entry is placed randomly into the PT-Rand region, 4KB of memory become readable. Hence, the attacker does not have to guess the correct address of a particular page table entry but only the start address of the page which contains the entry. As a consequence, the total randomization entropy available for PT-Rand is 28 Bit.

For a deterministic attack, the attacker has to manipulate a specific page table entry S that protects a specific function of the kernel. Alternatively, it might be sufficient for the attacker to corrupt an *arbitrary valid* entry A of the page table. However, it is not guaranteed that this modification will allow the attacker to compromise the kernel, thus, the attack success is probabilistic. Hence, we calculate the success probability that the attacker can correctly guess the address of the page which contains S . We denote this probability with $p(x)$ which depends on the number of pages, denoted by x that contain page table entries.

We can reduce the problem of calculating the success probability $sp(x)$ to a classical urn experiment without replacement and with three different colored balls: black, red, and green. The *black* balls represent the unmapped pages. The attacker loses the experiment by drawing a black ball (because accessing an unmapped page crashes the operating system). The *red* balls represent the valid pages, however, they do not contain the attacker's target page table entry S . The attacker is allowed to continue and draw another ball, as long as the attacker draws a red ball (access to a valid page). A *green* ball represents the page containing the page table entry S that the attacker aims to modify. With SG we denote the event that the attacker draws the green ball eventually without drawing a black ball (guessing the correct address of S without accessing an unmapped page). Hence, the probability of SG is the sum of the probabilities that the attacker draws the green ball in the first try plus the probability that the attacker draws the green ball after drawing the i -th red ball where $i \geq 1$. The resulting probability of SG is computed as follows:

$$\Pr[SG] = p(x) = \frac{1}{2^{28}} + \sum_{i=1}^x \frac{\binom{x}{i}}{\binom{2^{28}}{i}} * \frac{1}{2^{28} - i}$$

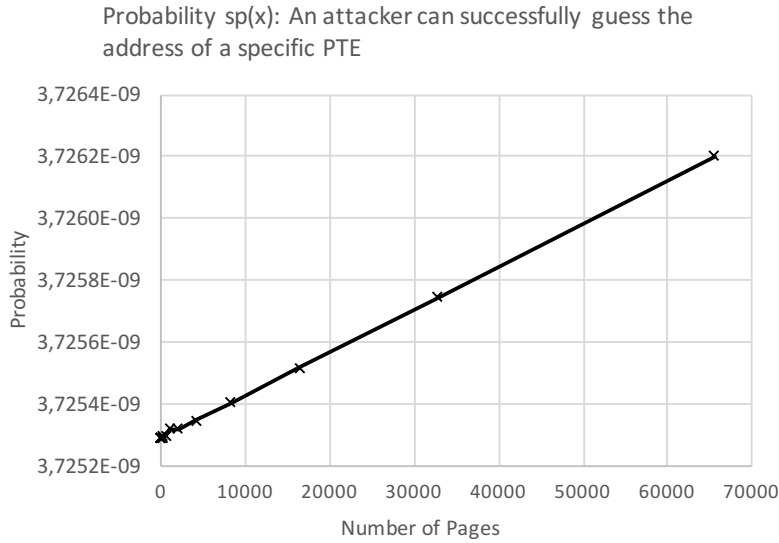


Figure 43: Probability for guessing attacks based on the number of mapped pages in the PT-Rand region.

Figure 43 plots the probability that the attacker can succeed in guessing a specific page table entry if up to 2^{16} memory pages for page tables are allocated. The graph shows that even if a high number of page table entries (PTEs) are allocated, the attacker's success probability is still very low ranging from 3.725×10^{-9} to 3.726×10^{-9} . We measured the number of page tables for a variety of different systems and configurations. For a normal desktop system, we observed that between 2,000 and 4,000 PTE pages were allocated. If we start a virtual machine up to 16,000 pages for PTEs are allocated. Lastly, our server (24 cores and 64GB RAM) running 9 virtual machines in parallel allocates up to 33,000 pages for PTEs. As shown in Figure 43, the probability grows linearly. Therefore, even if the attacker attempts to decrease the entropy by forcing the operating system to allocate more pages that contain page table entries⁸ the attacker's success probability is very low. Further, PT-Rand can prevent attacks on the entropy by limiting the amount of page tables to a number that will guarantee a user configurable amount of entropy.

For this reason, even if the attacker tries to decrease the randomization entropy by forcing PT-Rand to allocate a large amount of memory within the PT-Rand region, e.g., by spawning new processes, the success probability will not increase significantly before such an attack can be detected, e.g., by only allowing a fixed number of allocated pages.

Memory References

Memory disclosure is another severe threat to any kind of randomization scheme. For PT-Rand, we assume that the attacker can disclose any kernel data structure, and therefore, possible references to page tables. Hence, we obfuscate the references to page tables in all kernel data structures by substituting the virtual addresses with

⁸ the attacker can force the operating system to create new page table entries by starting new processes.

physical addresses. Note, there is no correlation between virtual and physical addresses. Therefore, the attacker does not gain any information about the real location of the page tables by disclosing obfuscated addresses. Since our modified memory allocator for page-table memory only returns obfuscated references, the attacker cannot access page tables by reading those pointers. The remaining potential source of leakage are functions that did not use our modified allocator. Recall, all functions that access the page tables now expect a physical address. Hence, if these functions receive a virtual memory address of a page table entry, they will automatically try to translate them using the randomization secret. The result is very likely an invalid address which will lead to a kernel crash.

Spilled Registers

As recently demonstrated in [45], even temporarily spilled registers which contain a security-critical value can compromise PT-Rand. To prevent *any* access to the debug register (DR3) that contains the randomization secret, we patched the Linux kernel code to never access DR3, i.e., DR3 cannot be accessed through any kernel API. Note that the CPU *does not* spill debug registers during interrupts [106]. Further, we prevent the compiler from writing the randomization secret to the stack by performing all computations in registers and never save or spill the result to memory. However, there might be cases, where a register that contains an intermediate value is spilled on the stack due to a hardware interrupt. In contrast to software interrupts, which we disable during page walks, hardware interrupts cannot be disabled. This opens a very small time window that may enable the attacker to use a concurrent thread to disclose register values, and potentially recover parts of the randomization secret. We performed preliminary experiments with a setting that favors the attacker to implement this attack, and did not succeed. Nevertheless, we are currently exploring two different strategies to mitigate such attacks. The first strategy is to further decrease the already small time window where register values could potentially be leaked. In particular, we envision to instrument the page table reads, by rewriting them with inline assembly, such that the de-obfuscated address is only present in the register for a couple of instructions. After accessing the page-table memory all registers that contain (intermediate values of) the randomization secret is set to zero. Alternatively, the second strategy ensures that the attacker cannot use a concurrent thread to access the stack of a victim thread that got interrupted and whose registers got temporarily spilled to memory. This can be achieved by using different page tables per kernel thread. Specifically, this allows us to assign stack memory per kernel thread which cannot be accessed by other (concurrent) threads. Therefore, even if intermediate values are spilled to memory, the attacker cannot leak them using concurrent threads. A simpler version of this technique, where the kernel uses a different page table per CPU, is already deployed in the grsecurity patch [202].

Real-world Exploit

We evaluated the effectiveness of PT-Rand against a set of real-world vulnerabilities. In particular, we use an information disclosure vulnerability in the Linux kernel to

bypass KASLR⁹, and a vulnerable driver which does not sanitize pointers provided by a user-mode application (CVE-2013-2595) to read and write arbitrary kernel memory. Based on these attack primitives, we develop an attack which allows us to execute arbitrary code in the kernel, despite having the kernel protected with state-of-the-art CFI for the kernel. The goal of our attack is to (i) change the memory permissions of a page that contains the code of a pre-defined kernel function to writable, (ii) overwrite the function with our shellcode, and (iii) finally trigger the execution of this function to instruct the kernel to execute our shellcode with kernel privileges.

To retrieve the KASLR offset, we use the aforementioned information disclosure vulnerability. The vulnerability allows the attacker to disclose the absolute address of a kernel function. Since we can determine the relative offset of this function to the start address of the kernel code section, we can compute the absolute address of the kernel after KASLR. Based on this address, we can compute the address of every function or global variable of the kernel since KASLR only shifts the whole kernel by a randomized offset during boot. In an offline analysis of the kernel image, we discovered a global variable that holds a reference to the `task_struct` of the initial process. The `task_struct` is a kernel data structure in which the kernel maintains information about each process, like id, name and assigned virtual memory. Specifically, it contains a pointer to the `mm_struct` which maintains information about the memory that is assigned to the process. Within this structure, we discovered a virtual memory pointer to the root of the page table of the corresponding process.

Using the arbitrary read capability and the 1:1 mapping, we traverse the page table to the entry that maintains the permissions for the system call `sys_setns`. Next, we set this page to writable and overwrite the beginning of `sys_setns` with our shellcode. In our proof-of-concept exploit, we re-write the function to elevate the current process' privileges to root. Naturally, other payloads are possible as well, like installing a kernel rootkit. After we modified the system call function, we set the corresponding page table entry again back to readable and executable, and invoke the system call to trigger the execution of our shellcode.

As explained in detail, this attack does not involve changing any code pointer. Hence, it resembles a data-only attack that cannot be mitigated by defenses against control-flow hijacking. However, after hardening the kernel with PT-Rand, this attack fails since we cannot reliably locate the correct page table entry for system call `task_struct`.

Side-channel Attacks

As stated in Section 4.4.3, preventing side-channel attacks is beyond the scope of this chapter. However, since side-channel attacks have the potential to undermine the security guarantees of PT-Rand, we will shortly discuss how these attacks work and how the kernel could be extended to prevent them.

Through side channels the attacker can disclose information about the kernel memory layout. In particular, the attacker discloses whether a kernel memory page is mapped.

⁹ . This vulnerability was silently fixed by the Linux kernel maintainers which is why there was no official CVE number assigned: <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=b2f739>

Hence, the attacker, in user mode, will attempt to read or write to a kernel memory page. Since kernel memory cannot be accessed by the user-mode programs such an attempt will result in an access violation. However, the time elapsing between the attempted access and the access violation depends on whether the page is mapped. Hund et al. [101] first demonstrated the feasibility of this attack by measuring the different timings the page fault handler needs to deliver an exception to the user mode to bypass kernel ASLR. Wojtczuk [232] improved this attack by using Intel’s Transactional Synchronization Extensions (TSX) which provides new instructions for hardware-aided transactional memory. The advantage of using TSX instructions to access kernel memory is that the faulting access does not invoke the page fault handler, and hence, allows to execute the previous attack of Hund et al. faster and with higher precision.

These timing-side channels exist because the user and kernel mode share the same address space, i.e., they use the same page tables. Hence, we can prevent such attacks by ensuring that the user and kernel mode use different page tables similar to Apple’s iOS [130].

Code-reuse attacks

PT-Rand is complementary to defenses against code-reuse attacks, like CFI [6, 78], CPI [118], or fine-grained randomization [51, 119]. We applied the open-source version of the CFI kernel protection for Linux RAP [166] to prevent the attacker from hijacking the control flow. Hence, the attacker cannot use code-reuse attacks like ROP to leak the randomization secret.

4.4.5.3 Performance

We rigorously evaluated the performance impact of PT-Rand using a variety of benchmarking suits on an Intel Core i7-4790 (3.6GHz) with 8GB RAM. In particular, we measure the impact on (1) CPU intensive applications using SPEC CPU, (2) start up time using LMBench, (3) on real-world applications using Phoronix, (4) and on JavaScript using JetStream, Octane, and Kraken.

On average, we observe an average run-time overhead of 0.22% for (1), 0.08% for (3), -0.294% for (4) and, 0.1ms increase in the start-up for (2). These results confirm that PT-Rand has no noticeable impact on the performance of the system, and hence, make PT-Rand a viable defense against page table attacks.

4.4.5.4 Robustness

To evaluate the robustness of PT-Rand we executed a large number of popular user-mode applications, and the three aforementioned benchmarking suites. We did not encounter any crashes during these tests, and all applications behaved as expected. To further stress test our implementation we executed the Linux Test Project (LTP) [123]. The LTP is comprised of different stress tests that can be used to evaluate the robustness and stability of the Linux kernel. We executed the most important tests under PT-Rand, and did not encounter any deviation in the behavior compared to the vanilla kernel.

Finally, we did not encounter any compatibility issues or crashes when combining PT-Rand with RAP [166].

4.4.6 *Discussion*

4.4.6.1 *Choice of 64-bit.*

The choice of 64-bit architectures is not a conceptual limitation. PT-Rand can be ported to 32-bit architectures. However, similar to ASLR, PT-Rand relies on the available randomization entropy which is known to be low for 32-bit systems [193]. Hence, we focused our efforts on hardening 64-bit-based architectures because nearly all commodity desktops and servers feature 64-bit CPUs. Even mobile devices are increasingly deploying 64-bit CPUs. As of 2013, Apple's iPhone embeds a 64-bit processor and iOS 9 runs exclusively on 64-bit processors. In a similar vein, Google runs 64-bit processors for their latest Nexus smartphone.

4.4.6.2 *Malicious Drivers.*

Our threat model does not consider injection of malicious drivers. These would allow the attacker to execute arbitrary code in kernel mode without requiring exploitation of a memory corruption vulnerability. As such, malicious drivers could access and leak the randomization secret. However, note that all modern operating systems support driver signing to prevent the loading of such malicious drivers thereby ensuring that the randomization secret is not leaked to the attacker.

4.4.6.3 *Physical Attacks.*

Similar to previous work [53], the main focus of this work is to prevent remote attacks against the kernel. As a result, attacks that rely on physical access to the victim system are beyond the scope of this work. For instance, several attacks in the past utilized special hardware (e.g., FireWire [181]) to create a snapshot of the physical memory [90]. Such snapshots can be analyzed by means of forensic tools to identify critical data structures such as the page tables in the case of PT-Rand. However, they require physical access to the RAM. Creating a memory snapshot remotely to detect the location of page tables is not feasible because the remote attacker has only access to virtual memory, i.e., linearly scanning virtual memory will eventually lead to a system crash since we move the page tables to a memory region where the majority of surrounding pages are not mapped.

Lastly, it is noteworthy to mention that PT-Rand does not depend on any specific operating system features and can be ported to other operating systems.

4.4.7 *Conclusion*

Exploitation of software is a pre-dominant attack vector against modern computing platforms. In particular, exploits against the kernel are highly dangerous as they allow the attacker to execute malicious code with operating system privileges. The

research community has introduced several classes of exploit mitigation techniques that significantly raise the bar of such attacks. However, these defenses build on the assumption that the attacker cannot alter the kernel's page tables which is the main place to manage access permissions of code and data memory. For the first time, we introduce a highly-efficient randomization technique that enables effective protection against page table corruption attacks for a contemporary Linux-based system. Our open-source solution, called PT-Rand, randomizes the location of all page tables, and obfuscates all references to the page tables without requiring extra hardware, costly hypervisors, or inefficient integrity checks. PT-Rand is a practical and necessary extension to complement existing mitigation technologies such as control-flow integrity, code randomization, and code pointer integrity.

4.5 RELATED WORK

In this section, we discuss related defensive research. We note that the recent effort in mitigating code-reuse attacks resulted in a large body of research, which goes beyond the scope of this dissertation. For example, automatic discovery of memory-corruption vulnerabilities by means of static and dynamic analysis [194, 235], or the development of new languages [110] are additional possible approaches to mitigate the risk of code-reuse attacks. Hence, we limit our discussion to research that is directly related to the defensive work of this chapter. In particular, we offer a detailed discussion of leakage-resilient software diversity and integrity-based defenses as an alternative defense against code-reuse attacks. We then briefly summarize existing general mitigations against data-only attacks, and end this section with an overview of kernel and page table mitigations.

4.5.1 *Leakage-Resilient Diversity*

Various research papers have been published on software diversity over the last two decades. We refer to Larsen et al. [119] for an overview.

Backes and Nürnberger [16] are the first to try to mitigate Just-in-Time Return-oriented Programming (JIT-ROP) attacks. Their Oxymoron approach uses the vestiges of x86 segmentation features to hide code references between code pages, which in turn prevents the recursive disassembly step in the original JIT-ROP attacks. However, as we demonstrate in Section 3.1.3 Oxymoron is vulnerable to indirect disclosure attacks.

The eXecute-no-Read (XnR) approach by Backes et al. [17] provides increased resilience against memory disclosure vulnerabilities by emulating eXecute-only Memory (XoM) on x86 processors. While the concept of XoM goes back to MULTICS [47], it is hard to support on x86 and other platforms that implicitly assign read permissions to executable pages. The XnR approach is to mark code pages *not present* so that any access invokes a page-fault handler in the operating system. If an access originates from the instruction fetcher, the page is temporarily marked present (and thus executable and readable), otherwise execution terminates. This prevents all read accesses outside a sliding window of recently executed pages.

Gionta et al. [80] demonstrate that XoM can also be implemented using a technique known as TLB Desynchronization on certain x86 processors. Whereas virtual addresses usually translate to the same physical address regardless of the type of access, the HideM approach translates reads and instruction fetches to distinct physical pages. This means that HideM, in contrast to XnR, can support code that embeds data in arbitrary locations. However, both mitigations can be bypassed using techniques we discussed in Section 3.2.3.1.

Gionta et al. [81] and Pomonis et al. [172] implement XoM for the kernel leveraging techniques we discussed in Section 4.2 and Section 4.1, respectively. Both schemes, however, do not implement comprehensive code-pointer protections, and hence, remain vulnerable to indirect disclosure attacks (cf. Section 3.1.3).

Lu et al. [124] present ASLRGuard, which aims to prevent the attacker from disclosing the code layout. Therefore, it uses a secure memory region, similar to a SafeStack [118], to store return addresses, and encryption for all other pointers, which are stored outside of the secure memory region. ASLRGuard relies on information hiding to protect the secure memory region, and to store the decryption key for pointers. However, encrypted code pointers can be reused for Counterfeit Object-oriented Programming (COOP)-like attacks (cf. Section 3.3).

Tang et al. [206] and Werner et al. [229] propose to prevent direct information-disclosure attacks through *destructive code reads* (DCR). DCR is based on the assumption that benign code will never read from the code section. Therefore, DCR intercepts memory-read operations to the code section, and overwrites the read bytes with random values before returning the original content. As a result, the attacker can learn the applied code randomization but she cannot leverage this information for a code-reuse attack because the code was replaced during the disclosure attack. However, as it turns out, these approaches are vulnerable to code-reload and code-inference attacks [197].

Pewny et al. [171] improve the original idea of destructive code reads. Contrary to previous work, the authors leverage static analysis to identify code and data within the code section. They utilize this information to enforce execute-only access for memory that was identified to contain code and read-only access for memory that contains data. Destructive code reads are only enforced for memory that could not be reliably identified as either code or data. Their technique drastically minimizes the attack surface, and prevents attackers from using known attack techniques to bypass destructive code reads.

Mohan et al. [143] present Opaque CFI (O-CFI), which is designed to tolerate certain kinds of memory disclosure by combining code randomization and integrity checks. Specifically, it tolerates code layout disclosure by bounding the target of each indirect control-flow transfer. Since the code layout is randomized at load time, the bounds for each indirect jump are randomized too. The bounds are stored in a small table, which is protected from disclosure using x86 segmentation. O-CFI uses binary rewriting and stores two copies of the program code in memory to detect disassembly errors, hence, it comes with a high memory overhead. Apart from the fact that O-CFI requires precise binary static analysis as its purpose is to statically resolve return addresses, indirect jumps, and calls, the attacker may be able to disassemble the code, and reconstruct (parts of) the control-flow graph at runtime. Hence, the attacker could dynamically disclose how the control-flow is bounded.

Another way to defend against information disclosure is live re-randomization, where the program is periodically re-randomized to invalidate any current code pointers, thereby preventing the attacker from exploiting any knowledge they gain of the program. Giuffrida et al. [82] describe the first implementation of this idea. However, even with very short randomization periods, the attacker may still have enough time for an attack [17, 61].

Bigelow et al. [21] propose an improved approach, TASR, which only re-randomize programs when they perform input or output operations that the attacker could

potentially exploit to disclose memory values. This approach requires that all code pointers are updated post-randomization, and rely on a modified C compiler to provide their locations. However, finding all code pointers in a C program is not always possible in the general case. The authors describe a set of heuristics and assumptions they depend upon to find the pointers, but real-world C code does not strictly comply with C's standard rules and often violates common sense assumptions about pointer use and safety [44].

Lu et al. [125] implement RuntimeAddress Space Layout Randomization (ASLR) which leverages a Pintool [126] to re-randomize the ASLR offsets after a process forks or clones itself. The `fork()` and `clone()` system call creates a new child process with an exact copy of the parent's address space, and is commonly used by server applications to handle client connections. The attacker can exploit this to brute force ASLR offsets by creating multiple requests and observe whether the child process crashes, which indicates wrong guess, or continues running [22]. Contrary to previous work, the authors use taint tracking to identify pointers.

Chen et al. [42] implement re-randomization for binaries. To track pointers, their approach relies upon indirection. Further, the authors deploy *honey-gadgets* [50] to counter guessing attacks.

Isomeron by Davi et al. [61] clones the code and switches between clones at each call site by randomly flipping a coin. If the coin comes up heads, an offset is added to the return address before it is used. Because the result of the coin-flip is stored in a hidden memory area, adversaries cannot predict how the return addresses in a return-oriented programming (ROP) payload will be modified by Isomeron.

4.5.2 Integrity-based defenses

The main focus of the defense part of this dissertation is on increasing the resilience of code randomization against memory-disclosure attacks to effectively mitigate code-reuse attacks. However, integrity-based defenses present a viable alternative. Therefore, we provide a brief overview about this direction of research to mitigate code-reuse attacks.

4.5.2.1 Control-flow Integrity

After Data Execution Prevention (DEP), Control-flow Integrity (CFI) [5, 6] is the most prominent type of integrity-based defense. Burow et al. [29] provide an excellent comparison of different implementations of CFI.

CFI constrains indirect branches in a binary such that they can only reach a statically identified set of targets. Since CFI does not rely on randomization, it cannot be bypassed using memory-disclosure attacks.

However, it turns out that the precise enforcement of control-flow properties invariably comes at the price of high performance overheads on commodity hardware. In addition, it is challenging (if not impossible) to resolve all valid branch addresses for

indirect jumps and calls. As a result, researchers have sacrificed security for performance by relaxing the precision of the integrity checks.

Coarse-grained CFI

Zhang et al. [237] present CCFIR, a coarse-grained CFI approach based on static binary rewriting that combines randomization with control-flow integrity. CCFIR collects all indirect branch targets into a *springboard* section and ensures that all indirect branches target a springboard entry. Unfortunately, the springboard is vulnerable to direct-disclosure attack (cf. Section 3.1.2), and hence, allows for a complete bypass.

Zhang and Sekar [239] present another coarse-grained CFI approach which relies on static binary rewriting to identify all potential targets for indirect branches (including returns) and instruments all branches to go through a validation routine. However, this mitigation merely ensures that branch targets are either call-preceded or target an address-taken basic block. Similar policies are enforced by Microsoft's security tool called EMET [138], which builds upon ROPGuard [73]. Microsoft's Windows 10 is the first operating system to deploy coarse-grained CFI [137]. A number of approaches have near-zero overheads because they use existing hardware features to constrain the control-flow before potentially dangerous system calls. In particular, x86 processors contain a last branch record (LBR) register which Pappas et al. [163], and Cheng et al. [43] use to inspect a small window of recently executed indirect branches. However, all these coarse-grained CFI policies give the attacker enough leeway to launch Turing-complete code-reuse attacks [31, 60, 83, 84, 185].

Davi et al. [58] and Pwenny and Holz [170] implement mobile-oriented CFI solutions based on binary rewriting and compilation of iOS apps respectively. Both implementations use static analysis augmented by either heuristics [58] or programmer intervention [170] to generate a control-flow graph (CFG) to restrict the program's control flow. This adds a high degree of uncertainty to the CFG's accuracy. A CFG that is too coarse-grained, i.e., places too few restrictions on the control flow, is easily exploitable by attackers, so the security of these defenses depends on the quality (granularity) of the generated CFGs.

Overwriting virtual tables (vtables) pointers (cf. Section 3.2.3.1) is a common attack technique to hijack the control flow of C++ applications. Hence, a number of recent CFI approaches focus on analyzing and protecting vtables in binaries created from C++ code [75, 173, 183, 221, 238]. While these approaches come with the advantage of being compatible with commercial off-the-shelf (COTS) binaries, the CFI policy is not as fine-grained as its compiler-based counterparts. Recently, Pawlowski et al. [164] introduced new techniques to recover C++ class hierarchies from stripped binaries, which can help to increase the precision of solutions that hard COTS binaries with CFI against code-reuse attacks.

As coarse-grained CFI fails to provide sufficient protection against code-reuse attacks, we now turn our attention to fine-grained CFI solutions.

Fine-grained CFI

Niu and Tan [153] demonstrate fine-grained compiler-based CFI implementation that is applied to individual modules, hence, supports dynamically linking shared libraries. Therefore, the authors extend the compiler to store control-flow information in each resulting binary, which is then used during run-time to extend the control-flow graph (CFG). Niu and Tan [154] extend their previous work to support Just-in-Time (JIT)-compiled code and C++, where they leverage a sandbox and double-mapping of the JIT-code memory to ensure that the attacker cannot modify the JIT-code during run time. Payer et al. [167] also utilize a sandbox for their fine-grained CFI implementation to protect return addresses. Finally, Niu and Tan [155] aim to increase the precision of their previous CFI implementations by leveraging points-to information which they collect during run time. Concurrently, van der Veen et al. [220] present a similar approach, albeit their precision is limited by the size of the Last Branch Record (LBR).

Mashtizadeh et al. [131] demonstrate fine-grained CFI using modern cryptography. Specifically, their CFI implementation relies on the AES instructions of recent x86 processors to protect pointers and uses the storage location as a nonce during encryption to reduce the ability of the attacker to reuse encrypted pointers in replay attacks.

PaX Team [166] implements fine-grained CFI for the Linux kernel. The author extends the GCC compiler to compute hashes from function signatures. The hashes are then used to ensure indirect function call only target functions with a matching function signature. The return addresses are protected through xor-based encryption scheme.

Another recent example of full-system CFI enforcement for an operating system kernel is KCoFI [53]. It securely stores the policies for safeguarding its virtualized guests inside a memory region that is only accessible through the hypervisor. However, this solution also comes with significant overhead of up to 200%, and requires a hardware-support for virtualization and deployment of a hypervisor.

A number of compiler-based CFI schemes focus on enforcing CFI for virtual function calls [26, 92, 111], and Tice et al. [215] generalize this idea to protect all indirect calls. The main idea is to extend the compiler to perform static analysis of the class hierarchy, and then instrument all virtual function calls to ensure that they can only target intended virtual functions.

Although these techniques have the advantage of adding only minimal performance overhead, they do not protect against attacks that use ret-terminated gadgets. Therefore, these defenses have to be combined with a shadow stack.

Dang et al. [56] provide an overview of different shadow stack implementations. Their results show that implementations with good performance are susceptible to memory-corruption attacks, and secure shadow stacks implementations cause a high performance overhead of up to 10% for common benchmarks.

Hardware-accelerated CFI

To overcome the performance challenges, researchers explore the possibility of extending the Instruction Set Architecture (ISA) to enforce CFI in hardware, or to leverage new hardware features to accelerate software-based CFI.

Arias et al. [11] and Sullivan et al. [204] implement CFI for the SPARC LEON3 processor, and report good performance results of below 2% avg. run-time overhead. Recently, Intel [107] announced that future Central Processing Units (CPUs) will provide native support for shadow stacks and coarse-grained CFI for call and jump instructions. Nyman et al. [156] introduce CFI for microcontrollers that are typically used in an Internet of Things (IoT) environment. Contrary to related work, their design takes interrupts into account which is important because microcontrollers commonly run bare-metal code.

Gu et al. [91], Ge et al. [79], and Liu et al. [122] leverage Intel's processor trace feature, which was originally intended for application profiling and debugging purposes, but also allows one to track the control flow, which in turn allows the enforcement of CFI.

4.5.2.2 *Software-Fault Isolation*

SFI isolates untrusted code so it cannot access memory outside the sandbox or escape confinement. SFI policies are typically enforced by inserting inline reference monitors [132, 184, 225].

Since reads are far more frequent than writes, some SFI implementations only sandbox writes and indirect branches. Google's NaCl implementation for ARM [187] eschewed load-isolation initially but support was later added [2] to prevent untrusted plug-ins from stealing sensitive information such as credit card and bank account numbers. NaCl for ARM uses a customized compiler and masks the high bits of addresses, and constrains writes and indirect branches. ARMor [28] is another SFI approach for ARM. It uses link-time binary rewriting to instrument untrusted code. This makes ARMor less efficient than compile-time solutions and the authors report overheads ranges from 5-240%.

Several hardware-based fault isolation approaches appeared recently. Zhou et al. [240] present ARMlock, which uses the memory domain support in ARM processors to create sandboxes that constrain the reads and writes, and branches of code running inside them with no loss of efficiency. While ARMlock prevents code from reading the contents of other sandboxes, it cannot support our use-case of preventing read accesses to code inside the sandbox. Santos et al. [182] use the ARM TrustZone feature to build a trusted language runtime (TLR); while this greatly reduces the TCB of open source .NET implementations, the performance cost is high.

4.5.2.3 *Memory Safety*

Code-Pointer Integrity (CPI) by Kuznetsov et al. [118], Szekeres et al. [205] aims to prevent pointer hijacking by storing code pointers, pointers to code pointers etc. in a safe region; all accesses to the safe region are instrumented to ensure the integrity of the pointers. Performance overhead is relatively small because CPI only needs to instrument a subset of memory operations. The critical issue is the protection of the safe region; on 64-bit Intel processors, segmentation is not available, thus CPI is forced to use information hiding. Unfortunately, the most efficient implementations of this defense can also be bypassed [68].

Nagarakatte et al. [148] extend the compiler to enforce spatial safety for C and C++. In a follow-up work, Nagarakatte et al. [149] added temporal safety as well. If combined, both defenses guarantee memory safety, however, they introduce a performance overhead of over 100% which is impractical for most use cases.

4.5.3 Data-only Defenses

To mitigate data-only attacks a number of data-randomization approaches have been proposed. Cadar et al. [30] and Bhatkar and Sekar [19] apply static analysis to divide data accesses into equivalence classes. Next, they instrument all data accesses to use an xor key per equivalence-class for reading and writing data from and to memory. This prevents the attacker from exploiting a memory-corruption vulnerability to access arbitrary data. However, the instrumentation of data accesses is expensive with up to 30% run-time overhead.

Castro et al. [33] aim to mitigate data-only attacks by enforcing *Data-flow Integrity* (DFI). The high-level idea is the same as in Control-flow Integrity. DFI relies on static analysis to infer a *data-flow graph* (DFG), and instrumentation of all read and write instructions to ensure that all data flows during run time are within the DFG. In a follow-up work, coined Write Integrity Test (WIT), Akritidis et al. [7] tackle the performance issues of DFI by reducing the precision of the DFG. Similar, to coarse-grained CFI, coarse-grained DFI would give the attacker enough leeway to perform attacks. Therefore, the authors additionally add CFI for forward edges to WIT.

4.5.4 Kernel and Page-Table Attack Mitigations

Several kernel defenses have been proposed that also protect the page table against malicious manipulations [14, 53, 77, 191, 226]. In general, existing approaches are based on a dedicated kernel monitor that enforces a set of pre-defined policies at run time, including integrity policies for page tables. To the best of our knowledge, PT-Rand is the first to use a randomization-based approach to defend against data-only attacks on page tables.

SecVisor [191] and HyperSafe [226] follow a hypervisor-based approach. SecVisor enforces $W \oplus X$ for the kernel space to ensure the integrity of the kernel code. This is done by using memory virtualization to allow only certain physical pages to be executable. SecVisor provides an interface to the kernel to allow new physical pages to be marked as executable. These requests are checked against a user-provided policy which is not further specified. HyperSafe protects its page tables by marking them read-only, and checks before updating the page tables if the update conforms to an immutable set of policies that should prevent malicious changes of page tables. Since the hypervisor maintains its own memory domain, virtualized guests cannot compromise its integrity by means of data-only attacks. However, the page tables maintained in the hypervisor itself can be compromised by the attacker. For instance, evasion attacks can be deployed to attack the hypervisor from a virtualized guest system [231]. Another practical shortcoming of hypervisor-based approaches is the incurred performance

overhead. SecVisor reports 14.58% average overhead (SPECInt) and HyperSafe 5% overhead (custom benchmarks). In contrast, PT-Rand only incurs 0.22% for SPEC CPU benchmarks. Some of the extra overhead of SecVisor and HyperSafe can be attributed to additional checks that go beyond table protection. However, the hypervisor itself will always add some extra execution overhead. In addition, these approaches rely on extra hardware features such as virtualization extensions.

SPROBES and TZ-RKP both leverage hardware trust anchors [15, 77]. In particular, both issue run-time checks for the kernel's memory management functions. These checks are executed inside the hardware-enabled secure environment ARM TrustZone. This secure environment cannot be tampered with by any other software. The overhead of TZ-RKP is up to 7.56%. In addition to the higher overhead, SPROBES and TZ-RKP rely on dedicated hardware trust anchors to protect page tables. SKEE implements similar run-time checks to SPROBES and TZ-RKP [14]. It utilizes the fact that ARM provides two registers for paging. This enables SKEE to isolate the run-time checks from the kernel. The overhead for protecting memory management varies between 3% and 15%. Policy-based approaches like HyperSafe [226] and SPROBES/TZ-RKP [15, 77] mark pages that contain the page table structures as read-only to prevent malicious modifications. However, when the operating system needs to update the page tables these defenses mark the corresponding pages temporarily writable which opens a time window in which the attacker can concurrently modify page-table entries on the same page. PaX/Grsecurity [202] provide a patch with various techniques to further harden the Linux kernel. Amongst others the patch aims to prevent information leaks, and randomizes important data structures at compile time. However, it does not deploy any techniques to explicitly prevent data-only attacks against the page table.

Windows 10 [108] recently released an update to randomize the base address, which is used to compute the address of page table entries. However, the randomized base address is not protected against information-disclosure attacks, which is why the attack we implemented in Section 4.4.5.2 will also work against Windows 10. In contrast, PT-Rand mitigates information-disclosure attacks by keeping the randomization secret in a register, which cannot be accessed by the attacker, and by obfuscating all pointers to the page tables.

4.6 SUMMARY AND CONCLUSION

In this chapter, we discussed how randomization-based defenses can be leveraged to efficiently mitigate code-reuse attacks. With Readactor we tackle the previously identified challenge of memory-disclosure attacks against randomization-based defenses. In particular, we leverage a very small hypervisor to enable memory virtualization, which allows us to enforce execute-only access for memory on desktop computers. We further modify the compiler to separate code and data, and to apply code randomization. This allows us to map the code section as execute-only, and hence, prevents direct disclosure attacks. To mitigate indirect disclosure attacks we introduce code-pointer hiding, which builds a layer of indirection and prevents the attacker from learning valid code addresses. Next, we presented techniques that allow one to implement execute-only memory for embedded devices that often do not feature hardware virtualization. Specifically, we split the address space into a code and data region, and instrument all read instructions of the binary to ensure that the address points to the data region. Finally, we presented the design of a linker wrapper that embeds compile-time information about the functions, as well as randomization code, to create self-randomizing binaries. Contrary to previous solutions, ours is highly practical, and is deployed in Tor Browser.

We then turned our attention to leveraging randomization to mitigate data-only attacks against the page table. Since the page tables are used to configure memory-access permissions, the attacker can easily compromise the system and bypass any other mitigations if she can tamper with the page tables. Previous work protects the page tables using an integrity-based approach. However, this is often comes with hardware dependencies, or has a negative impact on the overall performance of the system. Our randomization-based approach relocates the page tables into a large and unused memory region that provides enough entropy to prevent guessing attacks. Further, we protect all pointers to the page tables to ensure that the attacker cannot leverage an information-disclosure vulnerability to recover the randomization secret.

To conclude, randomization-based defenses can offer efficient and effective protection from memory-corruption-based attacks if combined with some form of leakage resilience.

DISCUSSION AND CONCLUSION

Memory-corruption vulnerabilities pose a serious threat to modern computer security because they allow the attacker to undermine other security primitives like access control, integrity checks, or secrecy. For example, the Heartbleed bug [133] is a simple memory-disclosure vulnerability, yet, it enabled attackers to leak the private SSL key of servers, or data of other users. Note that this bug only allowed to disclose memory. In other cases, memory-corruption vulnerabilities enable the attacker to corrupt data, which often leads to a complete take-over of the vulnerable system.

The main goal of this dissertation is to explore the limitations of state-of-the-art defenses against code-reuse attacks. Next, we leverage our new insights to design new defenses. We summarize the results of this dissertation in Section 5.1 and elaborate on future research directions in Section 5.2.

5.1 DISSERTATION SUMMARY

In Chapter 3, we presented memory-corruption attacks against code-randomization and Control-flow Integrity (CFI). Our attacks against randomization-based defenses show that the attacker can bypass any randomization scheme if she can repeatedly exercise a direct memory-disclosure vulnerability to read the content of the code section [196]. In addition, we show that preventing the attacker from reading the code section is not sufficient to harden code-randomization against disclosure attacks. By combining offline knowledge about the applied code randomization with run-time knowledge in form of leaked code pointers, the attacker can bypass code randomization [61]. While CFI is immune against disclosure attacks, it comes with its own weaknesses. Specifically, we show that the enforced policy of coarse-grained CFI gives the attacker enough freedom to conduct Turing-complete code-reuse attacks by leveraging virtual-function calls [186]. Furthermore, we find that the compiler can introduce security vulnerabilities in an effort to optimize the code for fine-grained CFI checks [45], which enables the attacker to bypass it as well. Finally, we demonstrate how the attacker can leverage data-only attacks to bypass code randomization and CFI by corrupting the intermediate representation of just-in-time compilers [72].

In Chapter 4 we turn our attention to mitigating code-reuse attacks. Therefore, we first evaluate eXecute-only Memory (XoM) as a potential primitive to mitigate information-disclosure attacks [51]. Our results show that XoM is suitable for the prevention of direct-disclosure attacks by setting the access permissions for the code section to execute only. Then, we leverage XoM to implement code-pointer hiding, which is an indirection for code pointers that effectively prevents indirect memory-disclosure attacks. Our approach leverages a small hypervisor to enable XoM because the regular memory permissions model does not distinguish between the read and execute

permission. Nowadays, x86-based Central Processing Units (CPUs) commonly support hardware virtualization, however, most embedded systems do not. Thus, we present a compiler-based approach that is tailored towards Reduced Instruction Set Computer (RISC)-based CPUs to implement XoM in software [27]. We also introduce the design of a fine-grained load-time randomization scheme that is reliable and adopted by real-world software [46]. We conclude our defense chapter with a novel defense against data-only attacks that target the page tables [62].

5.2 FUTURE RESEARCH DIRECTIONS

The problem of memory-corruption attacks has been known for over three decades, and is unlikely to be solved in the near future. The primary cause is that C and C++ are versatile languages, a large part of legacy software is based upon them, and it takes time to train programmers in alternative languages. Based on this assumption we believe that future research will focus on tolerating memory-corruption attacks.

Attack Surface Reduction.

The success of modern memory-corruption attacks often depends on degree the attacker can interact with the vulnerable application. For example, to perform the initial memory corruption the attacker often needs to bring the memory allocator into a certain state, e.g., by means of the heap feng shui technique [201]. Other times, the attacker needs to perform analysis during run time to adjust the payload [22, 61, 196], or exploit race conditions by spawning multiple threads [45]. Or, the attacker relies on a large code base to bypass control-flow integrity without violating the enforced policy [69, 186]. To summarize, many attacks rely on reusing the rich functionality of the vulnerable application. However, by isolating individual components, applications can reduce the functionality that is exposed to the attacker.

Software Fault Isolation (SFI) [2, 132, 187] provides isolation to execute untrusted code. Browsers, like Chrome and Edge, utilize a two-process approach. The first process runs with low privileges and executes the code base which is responsible for parsing webpages, executing JavaScript, and is likely to contain vulnerabilities. The second process runs with normal privileges and provides an interface for the first process to perform operations which require higher privileges, like reading files. As a consequence, the attacker, who is likely to compromise the first process, has to find another vulnerability in the second process, which has a much smaller attack surface. However, both approaches are very coarse-grained.

PartitionAlloc [209] (heap) and SafeStack [118] (stack) implement a more fine-grained approach by providing separated memory regions for allocating buffer objects and other objects because code that accesses the buffer is more likely to contain memory-corruption vulnerabilities. Hence, a buffer overflow cannot be exploited to overwrite pointers, which are present in other objects. Recently, Edge outsourced the native code generation for JavaScript to another process [141], thereby isolating the just-in-time compiler, which is likely to contain vulnerabilities due to its large code

base, from the part of the browser which is exposed to the attacker. Further, it enables Edge to enforce real Writable \oplus Executable memory. Note that other browsers do not fully utilize Writable \oplus Executable memory but either keep the just-in-time code memory as read-write-executable or re-map this memory from executable to writable and the other way around.

Isolating individual components greatly reduces the attack surface of an application. However, generic approaches are too coarse-grained to prevent the attacker from fully compromising the isolated component, and fine-grained approaches often require a manual redesign of the application. Therefore, we believe that designing primitives that enable fine-grained in-process isolation and (semi-) automatic approaches to isolate components of an application are important future research directions.

Formalization of Memory-corruption Attacks.

One limitation of current security research is the absence of a complete formal model for modern systems, which would allow one to verify the security properties of proposed mitigations. As a result, current code-reuse attack mitigations only defeat existing attacks but are commonly broken by previously unknown attacks. For example, Abadi et al. [6] provide a formal proof of the security properties of control-flow integrity. However, as we described in Section 3.2.3.2, the attacker can overwrite a return address, which is stored in user-mode memory, but used by the kernel to return from a system call. Similarly, Kuznetsov et al. [118] prove their security properties, however, the model used does not accurately represent the real system, and hence, one of their implementations was bypassed [68].

Building a complete formal model for modern systems is very challenging due to their high complexity. Nevertheless, we believe that the insights gained from the existing body of research will greatly support the creation of a formal model for systems that will allow a formal proof the security properties of code-reuse attack mitigations. As stated above, reducing the attack surface of application can be utilized to simplify such a formal model, and allow for further research.

ABOUT THE AUTHOR

Christopher Liebchen is a research assistant at the Technische Universität Darmstadt and the Intel Collaborative Research Institute for Secure Computing (Intel CRI-SC), Germany. In 2014, he received his M.Sc. in IT-Security from Technische Universität Darmstadt, Germany. His research focuses on memory-corruption attacks to bypass existing mitigations. He then uses these findings to further harden, or to design new mitigations.

PEER-REVIEWED PUBLICATIONS

1. Tommaso Frassetto, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. JITGuard: Hardening Just-in-time Compilers with SGX. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.
2. Orlando Arias, David Gens, Yier Jin, Christopher Liebchen, Ahmad-Reza Sadeghi, Dean Sullivan. LAZARUS: Practical Side-channel Resilient Kernel-Space Randomization. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2017.
3. Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory In *Proceedings of the 26th USENIX Security Symposium (USENIX Sec.)*, 2017.
4. Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.
5. Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, Hamed Okhravi Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.
6. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi Selfrando: Securing the Tor Browser against De-anonymization Exploits. In *Proceedings of the Annual Privacy Enhancing Technologies Symposium (PETS)*, 2016.
7. Ferdinand Brasser, Vinod Ganapathy, Liviu Iftode, Daeyoung Kim, Christopher Liebchen, Ahmad-Reza Sadeghi. Regulating ARM TrustZone Devices in Restricted

- Spaces. In *Proceedings of the 14th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.
8. Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Ahmad-Reza Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2016.
 9. Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
 10. Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, Michael Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
 11. Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz. Return to Where? You Can't Exploit What You Can't Find. In *Blackhat USA (BH US)*, 2015.
 12. Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.
 13. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.
 14. Luca Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, Fabian Monroe. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2015.
 15. Security Analysis of Mobile Two-Factor Authentication Schemes. Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, Ahmad-Reza Sadeghi. In *Intel Technology Journal, ITJ66 Identity, Biometrics, and Authentication Edition, Vol. 18* 2014.
 16. Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, Ahmad-Reza Sadeghi. On the (In)Security of Mobile Two-Factor Authentication. In *Financial Cryptography and Data Security (FC)*, 2014.

17. Kevin Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monroe, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: the More Things Change, the More They Stay the Same. In *Blackhat USA* (BH US), 2013.
18. Kevin Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monroe, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (S&P), 2013.
19. Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Ahmad-Reza Sadeghi. Over-the-air Cross-Platform Infection for Breaking mTAN-based Online Banking Authentication. In *BlackHat Abu Dhabi* (BH AD), 2012.

BIBLIOGRAPHY

- [1] ARM Compiler Software Development Guide v5.04, 2013.
- [2] Implementation and safety of nacl sfi for x86-64, 2015.
- [3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *12th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2005.
- [4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *9th International Conference on Formal Engineering Methods, ICFEM*, 2005.
- [5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2005.
- [6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.
- [7] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *29th IEEE Symposium on Security and Privacy, S&P*, 2008.
- [8] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7, 1996.
- [9] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 2000.
- [10] AMD. Intel 64 and IA-32 architectures software developer’s manual - Chapter 15 Secure Virtual Machine nested paging. <http://developer.amd.com/resources/documentation-articles/developer-guides-manuals>, 2012.
- [11] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFIX: Hardware-assisted flow integrity extension. In *54th Design Automation Conference, DAC*, 2015.
- [12] ARM. ARM architecture reference manual. http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf, 2015.
- [13] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.

- [14] A. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2016.
- [15] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014.
- [16] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *24th USENIX Security Symposium, USENIX Sec*, 2014.
- [17] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014.
- [18] S. Bhatkar and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *15th USENIX Security Symposium, USENIX Sec*, 2005.
- [19] S. Bhatkar and R. Sekar. Data space randomization. In *5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, DIMVA*, 2008.
- [20] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *13th USENIX Security Symposium, USENIX Sec*, 2003.
- [21] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [22] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy, S&P*, 2014.
- [23] Black Duck Software, Inc. Chromium project on open hub. <https://www.openhub.net/p/chrome>, 2014.
- [24] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *BlackHat DC*, 2010.
- [25] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ACM Asia Conference on Computer and Communications Security, ASIACCS*, 2011.
- [26] D. Bounov, R. G. Kici, and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2016.

- [27] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2016.
- [28] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization, CGO*, 2011.
- [29] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance, 2017.
- [30] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, 2008.
- [31] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *24th USENIX Security Symposium, USENIX Sec*, 2014.
- [32] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *25th USENIX Security Symposium, USENIX Sec*, 2015.
- [33] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2006.
- [34] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Electronic Voting Technology/Workshop on Trustworthy Elections, EVT/WOTE*, 2009.
- [35] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *17th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2010.
- [36] P. Chen, Y. Fang, B. Mao, and L. Xie. Jitdefender: A defense against jit spraying attacks. In *26th International Information Security Conference, IFIP*, 2011.
- [37] P. Chen, R. Wu, and B. Mao. Jitsafe: a framework against just-in-time spraying attacks. *IET Information Security*, 7(4), 2013.
- [38] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *15th USENIX Security Symposium, USENIX Sec*, 2005.
- [39] X. Chen and D. Caselden. CVE-2013-3346/5065 technical analysis. <http://www.fireeye.com/blog/technical/cyber-exploits/2013/12/cve-2013-33465065-technical-analysis.html>, 2013.
- [40] X. Chen, D. Caselden, and M. Scott. The dual use exploit: CVE-2013-3906 used in both targeted attacks and crimeware campaigns. <http://www.fireeye.com/blog/technical/cyber-exploits/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html>, 2013.

- [41] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [42] X. Chen, H. Bos, and C. Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *2nd IEEE IEEE European Symposium on Security and Privacy, Euro S&P*, 2017.
- [43] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2014.
- [44] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2015.
- [45] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [46] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. In *The annual Privacy Enhancing Technologies Symposium, PETS*, 2016.
- [47] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the MULTICS system. In *Joint Computer Conference, AFIPS*, 1965.
- [48] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *8th USENIX Security Symposium, USENIX Sec*, 1998.
- [49] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *13th USENIX Security Symposium, USENIX Sec*, 2003.
- [50] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *New Security Paradigms Workshop, NSPW*, 2013.
- [51] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy, S&P*, 2015.

- [52] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [53] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *35th IEEE Symposium on Security and Privacy, S&P*, 2014.
- [54] CVE Details. Linux kernel: Vulnerability statistics. <http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html>, 2016.
- [55] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 1989.
- [56] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *10th ACM Asia Conference on Computer and Communications Security, ASIACCS*, 2015.
- [57] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *6th ACM Asia Conference on Computer and Communications Security, ASIACCS*, 2011.
- [58] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *21st Annual Network and Distributed System Security Symposium, NDSS*, 2012.
- [59] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Asia Conference on Computer and Communications Security, ASIACCS*, 2013.
- [60] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *24th USENIX Security Symposium, USENIX Sec*, 2014.
- [61] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [62] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2017.
- [63] J. Drake. Stagefright: scary code in the heart of Android. <https://www.blackhat.com/us-15/briefings.html#stagefright-scary-code-in-the-heart-of-android>, 2015.

- [64] J. J. Drake. Exploiting memory corruption vulnerabilities in the java runtime. In *BLACK HAT ABU DHABI*, BH AD, 2011.
- [65] N. A. Economou and E. E. Nissim. Getting physical extreme abuse of intel based paging systems. <https://www.coresecurity.com/system/files/publications/2016/05/CSW2016%20-%20Getting%20Physical%20-%20Extended%20Version.pdf>, 2016.
- [66] J. Edge. Kernel address space layout randomization. <http://lwn.net/Articles/569635>, 2013.
- [67] S. Esser. iOS kernel exploitation. In *BLACK HAT EUROPE*, BH EU, 2011.
- [68] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.
- [69] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.
- [70] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *The 6th Workshop on Hot Topics in Operating Systems*, HotOS-VI, 1997.
- [71] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *New Security Paradigms Workshop*, NSPW, 2010.
- [72] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. Jitguard: Hardening just-in-time compilers with sgx. In *24th ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.
- [73] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012.
- [74] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Annual Computer Security Applications Conference*, ACSAC, 2009.
- [75] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference*, ACSAC, 2014.
- [76] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *25th Annual Network and Distributed System Security Symposium*, NDSS, 2016.

- [77] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *Mobile Security Technologies, MoST*, 2014.
- [78] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *1st IEEE IEEE European Symposium on Security and Privacy, Euro S&P*, 2016.
- [79] X. Ge, W. Cui, and T. Jaeger. Griffin: Guarding control flows using intel processor trace. In *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2017.
- [80] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Applications Security and Privacy, CODASPY*, 2015.
- [81] J. Gionta, W. Enck, and P. Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *IEEE Conference on Communications and Network Security, CNS*, 2016.
- [82] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *22nd USENIX Security Symposium, USENIX Sec*, 2012.
- [83] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *35th IEEE Symposium on Security and Privacy, S&P*, 2014.
- [84] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *24th USENIX Security Symposium, USENIX Sec*, 2014.
- [85] E. Göktas, R. Gawlik, B. Kollenda, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium, USENIX Sec*, 2016.
- [86] G. Gong. Pwn a nexus device with a single vulnerability. https://cansecwest.com/slides/2016/CSW2016_Gong_Pwn_a_Nexus_device_with_a_single_vulnerability.pdf, 2016.
- [87] Google. Chrome v8. <https://developers.google.com/v8/>, 2017.
- [88] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2017.
- [89] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *23rd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2016.

- [90] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *ACM Symposium on Cloud Computing, SoCC*, 2012.
- [91] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *7th ACM Conference on Data and Applications Security and Privacy, CODASPY*, 2017.
- [92] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *Annual Computer Security Applications Conference, ACSAC*, 2015.
- [93] J. Hiser, A. Nguyen, M. Co, M. Hall, and J. Davidson. ILR: Where’d my gadgets go. In *33rd IEEE Symposium on Security and Privacy, S&P*, 2012.
- [94] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *20th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2013.
- [95] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *International Symposium on Code Generation and Optimization, CGO*, 2013.
- [96] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity—program evolution redux. *IEEE Transactions on Dependable and Secure Computing*, PP, 2015.
- [97] M. Howard. Address space layout randomization in Windows Vista. http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx, 2006.
- [98] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *25th USENIX Security Symposium, USENIX Sec*, 2015.
- [99] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy, S&P*, 2016.
- [100] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *19th USENIX Security Symposium, USENIX Sec*, 2009.
- [101] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symposium on Security and Privacy, S&P*, 2013.
- [102] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. MAO – an extensible micro-architectural optimizer. In *International Symposium on Code Generation and Optimization, CGO*, 2011.

- [103] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 28 VMX support for address translation. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.
- [104] Intel. Software guard extensions programming reference. <https://software.intel.com/sites/default/files/329298-001.pdf>.
- [105] Intel. Intel 64 and IA-32 architectures software developer's manual, combined volumes 3A, 3B, and 3C: System programming guide. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf>, 2013.
- [106] Intel. Intel 64 and IA-32 architectures software developer's manual. <http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>, 2015.
- [107] Intel. Control-flow enforcement technology preview. <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>, 2016.
- [108] A. Ionescu. Owning the image object file format, the compiler toolchain, and the operating system: Solving intractable performance problems through vertical engineering, 2016.
- [109] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized nop insertion. In *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. 2013.
- [110] E. Jaeger and O. Levillain. Mind your language(s): A discussion about languages and security. In *IEEE Security and Privacy Workshops, LangSec*, 2014.
- [111] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2014.
- [112] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel TSX. In *23rd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2016.
- [113] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference, ACSAC*, 2006.
- [114] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*, 2014.

- [115] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *27th IEEE Symposium on Security and Privacy, S&P*, 2006.
- [116] B. Kollenda, E. Göktas, T. Blazytko, P. Koppe, R. Gawlik, R. K. Konoth, C. Giuffrida, H. Bos, and T. Holz. Towards automated discovery of crash-resistant primitives in binary executables. In *47th IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2017.
- [117] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. <http://users.suse.com/~krahmer/no-nx.pdf>, 2005.
- [118] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2014.
- [119] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy, S&P*, 2014.
- [120] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *30th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2007.
- [121] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, M. Franz, and U. Irvine. Subversive-c: Abusing and protecting dynamic message dispatch. In *USENIX Annual Technical Conference, ATC*, 2016.
- [122] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. Transparent and efficient cfi enforcement with intel processor trace. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, 2017.
- [123] LTP developer. The linux test project. <https://linux-test-project.github.io/>, 2016.
- [124] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [125] K. Lu, W. Lee, S. Nürnberger, and M. Backes. How to make aslr win the clone wars: Runtime re-randomization. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2016.
- [126] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40, 2005.
- [127] G. Maisuradze, M. Backes, and C. Rossow. What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses. In *25th USENIX Security Symposium, USENIX Sec*, 2016.

- [128] G. Maisuradze, M. Backes, and C. Rossow. Dachshund: Digging for and securing (non-)blinded constants in jit code. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2017.
- [129] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2011.
- [130] T. Mandt. Attacking the ios kernel: A look at "evasion". <http://www.nislabs.no/content/download/38610/481190/file/NISlecture201303.pdf>, 2013.
- [131] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [132] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *16th USENIX Security Symposium, USENIX Sec*, 2006.
- [133] N. Mehta. Heartbleed. <https://plus.google.com/+MarkJCox/posts/TmCbp3BhJma>, 2014.
- [134] M. Meissner. Tricks of a Spec master.
- [135] Metasploit. Metasploit. <http://www.metasploit.com/>.
- [136] Microsoft. Data Execution Prevention (DEP). <http://support.microsoft.com/kb/875352/EN-US/>, 2006.
- [137] Microsoft. Control flow guard. <http://msdn.microsoft.com/en-us/library/Dn919635.aspx>, 2015.
- [138] Microsoft. Enhanced Mitigation Experience Toolkit. <https://www.microsoft.com/emet>, 2015.
- [139] Microsoft. Chakracore. <https://github.com/Microsoft/ChakraCore>, 2015.
- [140] Microsoft. Hyper-V. <http://www.microsoft.com/hyper-v>, 2015.
- [141] Microsoft. Out-of-process jit support. <https://github.com/Microsoft/ChakraCore/pull/1561>, 2016.
- [142] Microsoft. Device guard. <https://pax.grsecurity.net/docs/pageexec.txt>, 2017.
- [143] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [144] Mozilla. W xor x jit-code enabled in firefox. <https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox>, 2015.

- [145] Mozilla Foundation. Ionmonkey. <https://wiki.mozilla.org/IonMonkey/Overview>, 2017.
- [146] MWR Labs. MWR Labs Pwn2Own 2013 write-up - kernel exploit. <http://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit>, 2013.
- [147] MWR Labs. Windows 8 kernel memory protections bypass. <http://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass>, 2014.
- [148] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *32nd Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2009.
- [149] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *ACM SIGPLAN International Symposium on Memory Management, ISMM*, 2010.
- [150] R. Naraine. Memory randomization (ASLR) coming to Mac OS X Leopard. <http://www.zdnet.com/blog/security/memory-randomization-aslr-coming-to-mac-os-x-leopard/595>, 2007.
- [151] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.
- [152] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *20th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2013.
- [153] B. Niu and G. Tan. Modular control-flow integrity. In *37th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2014.
- [154] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014.
- [155] B. Niu and G. Tan. Per-input control-flow integrity. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [156] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan. Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In *20th International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, 2017.
- [157] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *25th USENIX Security Symposium, USENIX Sec*, 2016.

- [158] Open Virtualization Alliance. KVM - kernel based virtual machine. <http://www.linux-kvm.org>.
- [159] OpenBSD. Openbsd 3.3, 2003.
- [160] Oracle Corporation. VirtualBox. <http://www.virtualbox.org>.
- [161] T. Ormandy. Cloudbleed. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1139>, 2017.
- [162] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy, S&P*, 2012.
- [163] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *23rd USENIX Security Symposium, USENIX Sec*, 2013.
- [164] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2016.
- [165] PaX Team. PaX address space layout randomization (ASLR). <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
- [166] PaX Team. RAP: RIP ROP. <https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf>, 2015.
- [167] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, DIMVA*, 2015.
- [168] O. Peles and R. Hay. One class to rule them all: o-day deserialization vulnerabilities in android. In *11th USENIX Workshop on Offensive Technologies, WOOT*, 2015.
- [169] Perception Point Research Team. Analysis and exploitation of a linux kernel vulnerability (cve-2016-0728). <http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/>, 2016.
- [170] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference, ACSAC*, 2013.
- [171] J. Pewny, P. Koppe, L. Davi, and T. Holz. Breaking and fixing destructive code read defenses. 2017.

- [172] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis. kr^x : Comprehensive kernel protection against just-in-time code reuse. In *12th European Workshop on Systems Security, EUROSEC*, 2017.
- [173] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [174] S. Quirem, F. Ahmed, and B. K. Lee. Cuda acceleration of p7viterbi algorithm in hmmer 3.0. In *Performance Computing and Communications Conference, IPCCC*, 2011.
- [175] S. Renaud. Technical analysis of the windows win32k.sys keyboard layout stuxnet exploit. http://web.archive.org/web/20141015182927/http://www.vupen.com/blog/20101018.Stuxnet-Win32k-Windows-Kernel_0Day-Exploit-CVE-2010-2743.php, 2010.
- [176] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *11th International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, 2008.
- [177] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15, 2012.
- [178] R. Rogowski, M. Morton, F. Li, F. Monroe, K. Z. Snow, and M. Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *2nd IEEE IEEE European Symposium on Security and Privacy, Euro S&P*, 2017.
- [179] R. Rudd, R. Skowrya, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2017.
- [180] J. Rutkowska and A. Tereshkin. IsGameOver() anyone? In *BLACK HAT USA, BH US*, 2007.
- [181] F. L. Sang, V. Nicomette, and Y. Deswarte. I/O attacks in Intel PC-based architectures and countermeasures. In *SysSec Workshop, SysSec*, 2011.
- [182] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2014.
- [183] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. Vtpin: Practical vtable hijacking protection for binaries. In *Annual Computer Security Applications Conference, ACSAC*, 2016.

- [184] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3, 2000.
- [185] F. Schuster, T. Tendyck, J. Powny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *17th International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, 2014.
- [186] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy, S&P*, 2015.
- [187] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *20th USENIX Security Symposium, USENIX Sec*, 2010.
- [188] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014.
- [189] F. J. Serna. CVE-2012-0769, the case of the perfect info leak. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf, 2012.
- [190] F. J. Serna. The info leak era on software exploitation. In *BLACK HAT USA, BH US*, 2012.
- [191] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *ACM SIGOPS Operating Systems Review*, 41, 2007.
- [192] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2007.
- [193] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *11th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2004.
- [194] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *37th IEEE Symposium on Security and Privacy, S&P*, 2016.
- [195] sinn3r. Here's that FBI Firefox exploit for you (cve-2013-1690). <https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690>, 2013.
- [196] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy, S&P*, 2013.

- [197] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *37th IEEE Symposium on Security and Privacy, S&P*, 2016.
- [198] Solar Designer. Getting around non-executable stack (and fix). <http://insecure.org/sploits/linux.libc.return.lpr.sploit.html>, 1997.
- [199] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [200] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2016.
- [201] A. Sotirov. Heap Feng Shui in JavaScript. In *BLACK HAT EUROPE, BH EU*, 2007.
- [202] B. Spengler. Grsecurity. *Internet [Nov, 2015]*. Available on: <http://grsecurity.net>, 2015.
- [203] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *4th European Workshop on Systems Security, EUROSEC*, 2009.
- [204] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *55th Design Automation Conference, DAC*, 2016.
- [205] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy, S&P*, 2013.
- [206] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [207] P. Team. PAGEEXEC. <https://pax.grsecurity.net/docs/pageexec.txt>, 2000.
- [208] P. Team. SEGMEXEC. <https://pax.grsecurity.net/docs/segmexec.txt>, 2002.
- [209] The Chromium Authors. Partitionalloc. https://chromium.googlesource.com/chromium/src/+/lkcr/base/allocator/partition_allocator/PartitionAlloc.md, 2013.
- [210] The Clang Team. Clang 3.8 documentation SafeStack. <http://clang.llvm.org/docs/SafeStack.html>, 2015.
- [211] The OpenSSL Project. Openssl. <https://www.openssl.org>, 2017.
- [212] The Tor Project. The tor browser. <http://www.torproject.org/projects/torbrowser.html>.

- [213] Theori. Chakra jit cfg bypass. <http://theori.io/research/chakra-jit-cfg-bypass>, 2016.
- [214] C. Tice. Improving function pointer security for virtual method dispatches. In *GNU Tools Cauldron Workshop*, 2012.
- [215] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *24th USENIX Security Symposium, USENIX Sec*, 2014.
- [216] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *14th International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, 2011.
- [217] Trusted Computing Group. Tpm 1.2 protection profile. <https://www.trustedcomputinggroup.org/tpm-1-2-protection-profile/>, 2016.
- [218] Ubuntu Wiki. Address space layout randomization (ASLR). <https://wiki.ubuntu.com/Security/Features#aslr>, 2013.
- [219] Unkown. Java 7 applet remote code execution. https://www.rapid7.com/db/modules/exploit/multi/browser/java_jre17_exec, 2012.
- [220] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [221] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *37th IEEE Symposium on Security and Privacy, S&P*, 2016.
- [222] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2017.
- [223] VMware, Inc. VMware ESX. <http://www.vmware.com/products/esxi-and-esx/overview>.
- [224] VUPEN Security. Advanced exploitation of internet explorer heap overflow (pwn2own 2012 exploit), 2012.
- [225] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th*, 1993.
- [226] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *31st IEEE Symposium on Security and Privacy, S&P*, 2010.

- [227] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *19th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2012.
- [228] Web Hypertext Application Technology Working Group (WHATWG). Chapter 10 - Web workers, 2015.
- [229] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Asia Conference on Computer and Communications Security, ASIACCS*, 2016.
- [230] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security Privacy*, 7, 2009.
- [231] R. Wojtczuk. Subverting the xen hypervisor. In *BLACK HAT USA*, BH US, 2008.
- [232] R. Wojtczuk. Tsx improves timing attacks against kaslr. <https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/>, 2014.
- [233] P. Wollgast, R. Gawlik, B. Garmany, B. Kollenda, and T. Holz. Automated multi-architectural discovery of cfi-resistant code gadgets. In *23rd European Symposium on Research in Computer Security, ESORICS*, 2016.
- [234] Xen Project. Xen. <http://www.xenproject.org>.
- [235] F. Yamaguchi. Pattern-based vulnerability discovery. <http://hdl.handle.net/11858/00-1735-0000-0023-9682-0>, 2015.
- [236] B. Zeng, G. Tan, and U. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *23rd USENIX Security Symposium, USENIX Sec*, 2013.
- [237] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy, S&P*, 2013.
- [238] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [239] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *23rd USENIX Security Symposium, USENIX Sec*, 2013.
- [240] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. Armlock: Hardware-based fault isolation for arm. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014.

- [241] D. D. Zovi. Practical return-oriented programming. Invited Talk, RSA Conference, 2010.

DECLARATION

Erklärung gemäß §9 der Promotionsordnung.

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Germany, February 2018

Christopher Liebchen